WEST

Freeform Search

Database:
US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: L2 and remote

Display: 40 **Documents in Display Format:** - **Starting with Number** 1

Generate: ○ Hit List [✓ Hit Count] ○ Side by Side ○ Image

Search | Clear | Help | Logout | Interrupt

Main Menu | Show S Numbers | Edit S Numbers | Preferences | Cases

Search History

DATE: Sunday, October 19, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
L4	l2 not L3	20	L4 Scanned all
L3	L2 and remote	54	L3 Scanned all
L2	L1 and ((@ad<20000421 or @rlad<20000421 or @prad<20000421)	74	L2
L1	(thread adj safe) and debug$4	107	L1

END OF SEARCH HISTORY

WEST

Generate Collection Print

L3: Entry 47 of 54 File: USPT Mar 28, 2000

DOCUMENT-IDENTIFIER: US 6042614 A
**** See image for Certificate of Correction ****
TITLE: System and method for a distributed debugger for debugging distributed application programs

Abstract Text (1):
A system and method for providing a distributed debugger system for a distributed target computer application are disclosed wherein the programmer/developer of the application can be at one host machine and wherein the application being developed makes use of objects and object implementations which may be located on a different host machine which is unknown to the programmer/developer. The system and method provides solutions to problems which are encountered in trying to debug a new application which is associated with the use of objects in a widely distributed, object oriented, client-server system. In a distributed object environment, requests and replies are made through an Object Request Broker (ORB) that is aware of the locations and status of objects. One architecture which is suitable for implementing such an ORB is provided by the Common Object Request Broker Architecture (CORBA) specification. The distributed debugger system disclosed herein is designed to function in a CORBA compliant distributed system.

Application Filing Date (1):
19980109

Brief Summary Text (3):
This invention relates to the fields of distributed computing systems, client-server computing and object oriented programming. More specifically, the invention is a method and apparatus for providing program developers and users the ability to debug target applications which may include programs or objects on distributed servers.

Brief Summary Text (5):
It is essential that a computer application developer be able to debug the application he is creating. This becomes an exceedingly difficult problem in an object-oriented, distributed processor environment. Such modern environments include applications which invoke objects developed by persons other than the application developer and may include implementations of the objects which are running on a processor remote from and unknown to the application developer. Nevertheless the application developer must have a way to debug the portions of the application that might reside on such remote and unknown processors.

Brief Summary Text (6):
For example, a distributed application is an application that comes in two or more parts. Those parts are often referred to as a client and its servers. Distributed applications have been in existence for many years and for just as many years program application developers have had the problem of debugging distributed applications. The typical method for debugging a distributed program is to start the client under a debugger and debug the client until one gets to a function that is in a server. If the developer is lucky, the server is already running on a known host. The developer then goes to the server host, identifies the server process, attachs a debugger to it, and continues the debugging session. If the server is not running yet, the developer must figure out how to get the server running and hope that whatever he did does not obscure the bug he/she is hunting. Once the server has been started the developer again attachs a debugger to it. Or the developer has to figure out how to interpose on the startup of the server so that he/she can attach a debugger to the server before anything interesting happens. This method is error prone, laborious, often confusing and tedious.

Brief Summary Text (10):
Another fundamental problem with prior art debuggers arises when one is faced with debugging an application which is implemented in a modern day distributed object system. Consider a Distributed Objects system of the type generally specified by the Object Management Group ("OMG"). OMG is a collection of over 500 companies which have agreed to certain specifications and protocols for such a distributed object system. The basic specification for this system is contained in the OMG Document Number 93.xx.yy Revision 1.2 dated Dec. 29, 1993 titled "The Common Object Request Broker: Architecture and Specification" (otherwise referred to as CORBA) which is incorporated herein by reference. Such CORBA compliant systems provide for building applications with pre-existing objects. Such applications can request the creation of an object and perform operations on that object. The creation and operations on objects are performed by servers for those objects. If such an application wants to create an object, it transparently utilizes a locator mechanism which finds a server known as a "factory" for that object. Similarly if such an application has a preexisting object, it transparently utilizes a locator mechanism to find a server that can perform operations on that object.

Brief Summary Text (11):
In such CORBA compliant systems there is a considerable amount of mechanism behind each object that allows the application programmer to use objects without knowledge of where the servers for the objects run. In the special circumstance where the developer of the client is also the developer of the server, arrangements can be made such that the programmer will know where the servers will run and what their names are. In general, however, the CORBA compliant system applications developer will be unable to locate the servers associated with his objects. Thus there exists a need to support debugging of objects used by applications regardless of whether the object is located in the same or remote process and regardless of where the object is. Moreover this debugging procedure should preferably create a "single process" illusion to the developer to allow the debugging of a large distributed application using a familiar debugging paradigm.

Brief Summary Text (12):
A major shortcoming of certain prior art debuggers is that they require a large overhead of supporting data and processes, usually comprising additional data generated by related compilers. Therefore a preferred embodiment for a remote debugger requires that object implementors should not have to do anything special to make their objects "debuggable" other than compile their servers so that symbolic information is generated. (In C and C++ using the -g compiler option.). However no additional behavioral or data abstractions on either servers or servant should be required lest the related overhead dominate the size and performance of fine grained objects. Similarly, another limitation of the prior art debugging systems is that they are linked to a specific type of target application and/or a specific compiler language. It is desired to have a distributed debugger be able to debug applications independent of the implementing language. That is, the preferred distributed debugger should not require any assumptions about the kinds of servers and objects it may operate on or operate with. The CORBA specification describes a variety of "Object Adapters" which may be used to service different kinds of object implementations. The desired distributed debugger should operate in an "Object Adaptor" independent manner if it indeed need not make any assumptions about the kinds of servers or objects it can operate with. Furthermore it is desired to have a distributed debugger that can ignore any boiler-plate code which the implementation of the CORBA compliant system might use to facilitate the operation of the system. "Boiler-plate" code refers to any non-programmer-generated code produced by the development environment and which is unknown to the developer. The distributed debugger should allow the developer to debug his system at the same functional level of abstraction at which they implemented the system.

Brief Summary Text (13):
The distributed debugger of the present invention, designated the "doeDebugger", provides an apparatus and method for accomplishing distributed debugging in a seamless, low-overhead, unencumbered manner, thereby permitting a developer to debug a distributed object oriented application with the illusion that he is debugging a "single process" application.

Brief Summary Text (15):
An apparatus and a method are disclosed whereby a client application can use a debugger on a local host, while at the same time being able to seamlessly debug an

application that involves objects and object implementations that may be running on unknown remote host computers.

Brief Summary Text (16):
A distributed bebugger system is disclosed for debugging a distributed target application system which may reside partly on a local host computer and partly on one or more remote host computers, the distributed debugger system having a debugger-GUI and one or more dbx engines which may reside on the local or a remote host computer, and a communications mechanism for use by the dbx engines and the debugger-GUI to talk to each other.

Brief Summary Text (17):
A further aspect of the invention claimed includes a distributed debugger system for debugging a distributed target application system which may reside partly on a local host computer and partly on one or more remote host computers, the distributed debugger system having a debugger-GUI and one or more dbx engines which may reside on the local or a remote host computer, and a communications mechanism for use by the dbx engines and the debugger-GUI to talk to each other, and having a dbx WrapperFactory mechanism for use by the debugger-GUI to create new dbx engines in remote host computers as necessary to provide the desired remote debugging support.

Brief Summary Text (19):
Another aspect of the present invention claimed includes a dbx engine for use in a distributed debugger system in debugging a distributed target application system which resides on a local host computer and on one or more remote host computer units, the dbx engine comprising a dstep mechanism for ignoring non-programmer generated code, a testing mechanism for identifying such non-programmer generated code (also called IDL generated code), a mechanism for setting remote breakpoints in sections of the target application system, a GetImplementation mechanism (also called a "find server" mechanism) for identifying the host ID and process ID (pid) of a server which implements a called object, and a multiple dbx engine synchronizer mechanism for permitting dbx engines to communicate with each other.

Brief Summary Text (20):
Also claimed in this application are methods for producing a dbx engine having the characteristics described above, as well as methods for producing a distributed debugger system as described above.

Drawing Description Text (7):
FIG. 5 illustrates the SPARCworks debugger.

Drawing Description Text (8):
FIG. 6 illustrates the relationship between a client application, an object reference and the SPARCworks debugger.

Drawing Description Text (11):
FIG. 9 illustrates a doe Debugger configuration.

Detailed Description Text (7):
The following disclosure describes a system and method for debugging a distributed computer application wherein the programmer/developer of the application is at one host machine and wherein the application being developed makes use of objects and object implementation which may be located on a different host machine which is unknown to the programmer/developer. The system and method provides solutions to problems which are encountered in trying to debug a new application which is associated with the use of objects in a widely distributed, object oriented, client-server system. Distributed objects can be object clients or object servers depending on whether they are sending requests to other objects or replying to requests from clients. In a distributed object environment, requests and replies are made through an Object Request Broker (ORB) that is aware of the locations and status of objects. One architecture which is suitable for implementing such an ORB is provided by the Common Object Request Broker Architecture (CORBA) specification. The implementation described, while it may be used in any relevant context, is an extension to the Distributed Object Environment ("DOE") system of Sun Microsystems, Inc. DOE is Sun's present implementation of the CORBA architecture. However, no specific knowledge of the DOE system is required by those skilled in these arts to understand and implement the process and system described in this disclosure.

Detailed Description Text (8):

The present invention discloses systems and methods for creating and using a doeDebugger which can permit a programmer/developer to ignore the fact that an object invoked by his application may be implemented remotely, and does not require him to do anything special because of a remote implementation, and which does not incur an inordinate overhead load, which is secure and which allows the programmer/developer to debug his system at the same functional level of abstraction at which he implemented the system. Alternate implementations would also provide some language independence.

Detailed Description Text (20):
III. The Distributed Debugger--How To Make It

Detailed Description Text (21):
Referring now to FIG. 5, The SPARCworks debugger system is depicted. The SPARCworks debugger (hereinafter the "Debugger") is an integrated component of the SPARCworks toolset produced by Sun Microsystems, Inc. which includes an Analyzer, a dbx engine, a FileMerge tool, a Maketool, a Manager and a SourceBrowser. The Debugger is more fully described in the publication titled "Debugging a Program" published by SunSoft as Part No. 801-7105-10 dated August 1994 and which is fully incorporated herein by reference.

Detailed Description Text (22):
Referring now to FIG. 5 the Debugger comprises a sophisticated window-based tool 72 (the Debugger Graphical User Interface (GUI)) that interfaces with the dbx engine 76. The dbx engine 76 is an interactive, line-oriented, source-level, symbolic debugger. The dbx engine 76 permits one to determine where a target program crashed, view the values of variables and expressions, set breakpoints 78 in the code, and run and trace a target program. During program execution, the dbx engine 76 obtains detailed information about target program behavior and supplies the Debugger GUI 72 with this information via a ToolTalk communications protocol 74. The dbx engine 76 relies on debugging information a compiler generates using the compiler option -g to inspect the state of the target process. By default on Solaris 2.x, the current Sun Microsystems, Inc. operating system environment, debugging information for each program module is stored in the module's .o file. In the preferred embodiment on Solaris 2.x, the dbx engine 76 reads in the information for each module as it is needed. In addition to the "set breakpoint" 78 capability, the dbx engine 76 has a "single step" 80 capability as well as many other 82 features which are described in more detail in the above reference publication "debugging a Program." The "step" 80 feature allows the programmer/developer to single-step through target program code one line at a time at either the source or machine-language level; step "over" or "into" function calls; step "up" and "out" of a function call arriving at the line of the calling function line (but after the call). There are three type of breakpoint 78 commands;

Detailed Description Text (23):
(1) stop type breakpoints--If the target program arrives at a breakpoint created with a stop command, the program halts and optionally issues one or more debugging commands. Another debugging command must be issued in order to resume the target program.;

Detailed Description Text (24):
(2) when type breakpoints--the target program halts and the dbx engine issues one or more debugging commands and then the target program continues.; and

Detailed Description Text (26):
In a non-distributed system, a typical configuration of the Debugger is shown in FIG. 6. Therein a host machine 92 is shown containing the debugger GUI 94 connected to a dbx engine 98 by a ToolTalk communications link 96, with the dbx engine 98 linked to a client (target program) 100 which is further connected to additional target program application code (server) 102. In a distributed system the programmer/developer is faced with one wherein the situation is more like the one shown in FIG. 7. FIG. 7 shows multiple clients with multiple servers on multiple hosts. A programmer/developer using the Debugger 114 on the red host 112 to debug client 1 116 may find that the client 1 116 performs an operation on an object which can be performed by the server 118 on the red host 112 or by the server 124 on the blue host 122 and the programmer as the developer of client 1 116 does not know which server will be used for the execution of a call. In addition, whichever server is used by client 1 116 may also be used by client 2 128 on the white host 130. In a CORBA compliant distributed system such as DOE (the preferred embodiment of the

present invention) there is a considerable amount of mechanism behind each object that allows the application programmer to use objects without knowledge of where the servers for the objects run. This seriously hampers the debugging of distributed applications under such circumstances. Furthermore all DOE servers are multithreaded servers, in which it is possible that the server will be servicing multiple requests from different clients simultaneously. Accordingly, the doeDebugger of the present invention is a solution to many of the debugging problems created by the DOE distributed environment.

Detailed Description Text (27):
Before describing the modifications to the Debugger necessary to create the doeDebugger of the present invention, it is helpful to look briefly at the boiler plate mechanism code used by DOE to permit objects to communicate with each other in a CORBA compliant environment. Referring to FIG. 8 a view of the division between used code and underlying DOE mechanism is shown.

Detailed Description Text (29):
For example in FIG. 8, the user has defined a function foo in IDL. The IDL compiler generates a stub function foo 148, 150 which causes a message to be created 152 that is sent to the server. On the server side, the IDL generated code 160, 162 takes the message from the client and converts it into a call to the function foo 164 that is provided by the programmer that implements the functionality of foo. All of this underlying code 148, 150, 152, 158, 160 and 162 is code that a programmer/developer does not want to debug and does not want the debugger to look at in a normal debug session. Consequently it is desired to allow a distributed debugging session to ignore all of this underlying code as far as the programmer/developer is concerned.

Detailed Description Text (30):
FIG. 9 depicts the modifications and extensions required to convert the SPARCworks Debugger into the doeDebugger which is the present invention. The actual extensions are packaged in a shared library "libdoeDebugger.so" 204. The fundamental extensions required include the following:

Detailed Description Text (32):
a "remote surrogate code test" mechanism 208

Detailed Description Text (33):
a "Remote Breakpoint" setting mechanism 210

Detailed Description Text (37):
In addition to these extensions, the modifications to the Debugger-Gui and the dbx engine to support the doeDebugger operation included:

Detailed Description Text (39):
ability in the debugger-GLTI to focus it on a particular dbx engine; and

Detailed Description Text (40):
the ability to get a list of all active dbx engines from the debugger-GUI

Detailed Description Text (42):
The "dstep" command will be used by the programmer/developer to seamlessly step into the implementation of a given function, regardless of where in the distributed system the function's implementation actually resides. "dstep" works by first issuing a normal dbx "step" command. The standard "step" command continues the execution of the process being debugged (the debugee) from the current source line to the next source line of the debuggee. When the current line is at the point of invocation of a function, the next source line is the first source line in the function being called. In order to extend the semantics of the step command to debugging of distributed applications, when the extended step enters a function such as foo on the client in FIG. 8 (point A 146), execution should stop at the first line in the implementation of foo in the server (point B 166). The extended step command (which will be referred to as the "dstep" command) will operate as the standard step command except in the following two situations:

Detailed Description Text (44):
When the "dstep" command is executed on the return from a function invoked in the server, the next source line will be the source line in the client after the function invocation that resulted in the remote invocation. In terms of the DOE clients, the "dstep" command starts its special functions when the user steps into

the DL generated code. In terms of the DOE servers, the "dstep" command starts its special function when the user returns through the IDL generated code.

Detailed Description Text (45):
It is also necessary to shut down the doeDebugger transparently. Since the user does not know what has been done to support the remote debugging, he/she should not be expected to undo it.

Detailed Description Text (46):
After the "dstep" command is executed, the doeDebugger tries to determine if the current function is "remote surrogate code". "Remote surrogate code" is that code responsible for causing the remote invocation of the DL operation. (that is, items 148, 150 in FIG. 8). Currently in the DOE system all of the "remote surrogate code" is generated by the IDL compiler.

Detailed Description Text (48):
As would be expected, the name of the client side function being called is the same as the name of the server side function that implements that function. In the example in FIG. 8, this just means that the programmer calls function "foo" on the client side and expects to get the function "foo" that he/she wrote on the server side. A server may be servicing many clients and therefore there may be many invocations of that function "foo" occurring in the server. The invocation of interest in the server is identified by determining which thread in the server was servicing the function call of the client being debugged. Two functions (one on the client side and one on the server side) were added to the message passing layer in DOE to aid in the identification of the specific thread in the server.

Detailed Description Text (54):
When the doeDebugger steps into a server for the first time, it must start a new dbx-engine on the server. This process is part of the "IdentifyRemote Function" process which was added as an extension to identify remote dbx engines, start/create a remote dbx engine using the facilities of a dbx WrapperFactory object. How this is done is now described with reference to FIG. 14. FIG. 14 depicts a local host 520 having a debugger-GUI 502, a dbx engine 504, a helper process 506, a client side wrapper server 510 and a client 508. Also shown are a remote host 522 containing a dbx engine 512, a helper process 514, a server side wrapper server 518 and the server (implementation of the called function) 516. Initialization of a new dbx engine including connection to the debugger-gui and attachment to the server is accomplished by the client side dbx engine 504 making a call to create a new dbx-engine.

Detailed Description Text (55):
The dbx engine 504 on the local host 520 creates the dbx engine 512 on the remote host 522 by means of a request to the wrapper server 510 on the local host 520 via the helper process 506 on the local host 520. The helper process 506 is a DOE application that communicates with the wrapper server 510. It is necessary because dbx engines themselves are not multi-thread safe (MT safe) and cannot be made into a DOE application (all DOE applications are inherently multithreaded). The dbx engines access the services of the wrapper server through the helper process. The wrapper server 510 on the local host 520 sends a message to the wrapper server on the remote host 522 requesting that a dbx engine be created on the remote host 522 and be instructed to attach to the server 516. The request by the dbx engine 504 on the local host 520 to create the dbx engine on the remote host 522 does not complete until the dbx engine 512 on the remote host 522 is fully started. The wrapper server on the remote host 522 forks and exec's the new dbx engine 512 on the remote host 522 and then waits for the dbx engine 512 to either terminate or send it (wrapper server 518) a message that indicates that the dbx engine 512 is fully started. The wrapper server 518 on the remote host 522 creates two threads. One thread waits for the forked child (dbx engine 512) to terminate. The other thread waits for a message from the dbx engine 512 that it is fully started. When one of those threads resumes, it destroys the other thread and sends the appropriate reply back to the wrapper server 510 on the local host 520 which in turn completes the request for the creation of the dbx engine 512 on the remote host 522 and returns the appropriate status value (creation succeeded or failed) to dbx engine 504 on the local host.

Detailed Description Text (56):
After the new dbx-engine 512 on the server has started (as described above), the dbx-engine 504 on the client sends a message to the new dbx-engine 512 to set the appropriate breakpoint in the server 516. The dbx-engine 504 on the client cannot

send the breakpoint message to the dbx-engine 512 on the server until that dbx-engine 512 has been full started (i.e., it has gone through its initialization including connection to the debugger-gui and has attached to the server).

Detailed Description Text (58):
The description of the steps performed during a "dstep" refers to sending a command to the dbx-engine attached to the server to set a breakpoint in the server. The command contains sufficient information to set the breakpoint on the correct thread in the server. When the step enters the trigger function, the information does not yet exist which can uniquely identify the thread in the server that will service the call resulting from the call of the trigger function. The actual mechanism is that an event is set in the transport layers at a point where an identifier for the request (the request id) resulting from the call of the trigger function is available. That event sends a command to the dbx-engine on the server. The message contains the request id, host name of the client, and interprocess address of the client and that uniquely identifies the request The message sent to the dbx engine on the remote host results in a breakpoint set set in the message passing layer in the server. That breakpoint checks for a match of request id, host name of client, and client interprocess address and when a match is found, the thread that makes the match will be the thread that services the request that originated with the trigger function in the client. A breakpoint is then set for that thread on entry to the function and the server is then continued.

Detailed Description Text (61):
As described above with reference to FIG. 14, a message is sent from the dbx engine 504 on the local host 520 to the dbx engine 512 on the remote host 522 to set a breakpoint in the server 516. After that breakpoint message is sent, the client 508 is continued so that the remote invocation proceeds from the client 508 to the server 516. There is however, no guarantee that the breakpoint message is received and processed by the dbx engine 512 on the remote host 522 before the remote invocation reaches the server 516. To guarantee that the breakpoint is actually set before the remote invocation takes place, the dbx engine 504 on the local host 520 stops and waits for a message from the dbx engine 512 on the remote host 522. Once the dbx engine 512 on the remote host 522 sets the breakpoint in the server 516, it sends a message to the dbx engine 504 on the local host 520 to continue the client 508.

Detailed Description Text (63):
The changes necessary to permit one dbx engine to communicate to another dbx engine included changes to both the debugger-GUI and dbx engine such that a specific ToolTalk message (referred to as the "rcmd" message) could be passed from the dbx engine to the debugger-GUI for the purposes of having a message sent to another dbx engine (referred to as the target dbx engine). The debugger-GUI accepts commands from the user and forwards them to specific dbx engines. The "rcmd" message contains the name of a host machine where the target dbx engine is running, the process identifier (pid) of the process being debugged by the target dbx engine and the message for the target dbx engine. The debugger-GUI maintains a list of dbx engines and this list contains the name of the host where the dbx engine is running and the pid of the process being debugged. When the debugger-GUI gets the "rcmd" messages, it searches its list of dbx engines for a dbx engine that is running on the named host machine and is debugging a process with the given pid. The message for the target dbx engine is then delivered to that dbx engine.

Detailed Description Text (64):
The changes necessary to permit the debugger-GUI to focus on a particular dbx engine included changes to both the debugger-GUI and dbx engine such that a specific ToolTalk message (referred to as the "attention" message) could be passed from the dbx engine to the debugger-GUI for the purposes of having the debugger-GUI change the focus of its attention to the sending dbx engine. The debugger-GUI maintains displays that relate to a specific dbx engine. For example, the source program for the process being debugged by a particular dbx engine is displayed by the debugger-GUI. The debugger-GUI has one set of displays and can show the information from one dbx engine at a time. The "attention" message tells the debugger-GUI to change its displays from the information for the current dbx engine to the information of the dbx engine sending the "attention" message.

Detailed Description Text (65):
The changes necessary to permit one to get a list of all active dbx engines from the debugger-GUI included changes to both the debugger-GUI and dbx engine such that a

specific ToolTalk message (referred to as the "get dbx engines" message) could be passed from the dbx engine to the debugger-GUI for the purposes of having the debugger-GUI send back a list of the host names where each dbx engine was running and process identifier (pid) of the process being debugged by that dbx engine. The debugger-GUI maintains a list of all dbx engines which includes the name of the host where the dbx engine was running and the pid of the process being debugged by that dbx engine. When the debugger-GUI receives the "get dbx engine message", it extracts the name of the host and the pid of the process being debugged for each dbx engine and sends that back to the dbx engine sending the get dbx engines message.

Detailed Description Text (66):
IV. The Distributed Debugger--How To Use It

Detailed Description Text (67):
Having described the changes/extensions to the SPARCworks debugger system necessary to create the doeDebugger of the present invention, the method of using the doeDebugger is now described.

Detailed Description Text (68):
Referring now to FIG. 10, the doeDebugger operation 220 is described. To begin, a programmer/developer on a local machine starts doeDebugger 222. The target program is indicated 224 and a "dstep" command is specified for a desired function 226. The doeDebugger executes a standard "step" command 228 and the doeDebugger attempts to determine if the target implementation is local or remote 230. Recognizing that the target is remote (by recognizing the "remote surrogate code" generated by the IDL), the "find server" function is executed 232 to find the host id and pid of the target implementation. The local dbx engine then issues a command to create a dbx engine in the found host 234 and blocks and waits for a response. The found host determines if there is a dbx engine connected to the target implementation 236. If there is already a dbx engine running 250 the server on the found host sends a return message to the calling client side dbx engine that a dbx engine is running. 252 The client side dbx engine receives the message and unblocks 254. The client side dbx engine then sends a message to the dbx engine on the server to set a temporary breakpoint in the designated function. 262 (in FIG. 11). Continuing in FIG. 11, the server dbx engine executes the command to set breakpoint in the target function 264 and saves the return trigger stack pointer 266. The server dbx engine "continues" the target implementation 268. Subsequently the target implementation hits the designated breakpoint 270 and the server dbx engine services the breakpoint and sends a message to the debugger-GUI on the client host to focus on the server dbx engine 272. The programmer using the debugger-GUI is now able to debug the remote function as if it were on the client host 274. Thereafter the system checks to see if the debug session is finished (i.e. quit command received) 276 and if not 278 the debug session continues 282. If the quit command was received 280 the remote dbx engine quits and detaches from the target process 284 and exits the session 286.

Detailed Description Text (70):
The "Remote dbx engine Create and attach" process depicted in block 246 of FIG. 10 is now described in more detail with reference to FIG. 12. In FIG. 12, the "create" process 302 is initiated and the client side dbx engine calls the local helper process 304. The local helper process issues a "create" command to the dbx WrapperFactory object 306. The dbx WrapperFactory object sends a message to the dbx WrapperFactory implementation on the found host 308 which executes the "create" command 310. The dbx WrapperFactory implementation does a "fork" and "exec" for a dbx engine 316 and waits for a message from the newly created dbx engine 318. If the new dbx engine does not get fully stated for some reason 324 a "Failed.sub.-- to.sub.-- start" message is returned 326 and the dbx WrapperFactory implementation exits 320. If the new dbx engine does get fully started 322 the new dbx engine is directed to attach to the target function 328. If the new dbx engine is not able to attach 334 a message "Failed.sub.-- to.sub.-- attach" is returned 336 and the dbx WrapperFactory implementation exits 320. If the new dbx engine is able to attach 332 a message "Attached.sub.-- and.sub.-- running" is returned and the dbx WrapperFactory implementation exits 320. It should be noted that "failure to attach" to the target server may result from a server which has its own permission to attach requirements, in which case other mechanisms for attaching may be required.

Detailed Description Text (71):
The "Remote dbx engine Quit and Detach" process depicted in block 284 of FIG. 11 is now described in more detail with reference to FIG. 13. In FIG. 13, the "Quit" process begins 402 with the programmer/developer issuing a "Quit" command to the

debugger-GUI which passes the command to the local dbx engine 404. The local dbx engine sends a "QuitSession" message via the helper process to the dbx WrapperFactory to quit the debugging session 406. The "QuitSession" command causes the dbx WrapperFactory objects on each of the participating hosts to send a signal 408 to each dbx engine that is part of the debugging session. 410 Each dbx engine has a signal handler for the sent signal which checks to see if the signal is being sent from the dbx WrapperFactory and if it is, the dbx engine detaches from the process it is debugging 412 and quits. Alternative embodiments could include additional steps such as, detaching the dbx engine process from its target function 414, and returning a message like "dbx.sub.-- debugger.sub.-- detached.sub.-- and.sub.-- deleted" to the client side dbx engine 416.

Detailed Description Text (73):
Although the present invention has been described with reference to particular operating systems, program code mechanisms, and object and object reference definitions, it will be appreciated by one skilled in the art that the present invention may be implemented in any one of a number of variations within a given operating environment, or in different operating system or object system environments. Similarly, particular client and server configurations or combinations illustrated in the figures are only representative of one of many such configurations of clients and servers and object and sub-object relationships which may use the present invention. Moreover, it will be understood that the figures are for illustration only and should not be taken as limitations on the invention. Some additional combinations of the remote dbx engine with a client side debugger-GUI with other functions include the combining of the dbx engine with a Graphical User Interface ("GUI") agent that provides a friendly user interface to the target object; the combining of the remote dbx engine with an artificial intelligence agent which modifies remote requests based upon the user's known preferences; the combining of the remote dbx engine with a caching program that caches answers to remote requests; the combining of the remote dbx engine with a teleconferencing application that merges the inputs from several users and sends them to the target; or the combining of a remote dbx engine with a number of audio and video accessing agents in a multimedia system. These possible dbx engine and debugger-GUI combinations are not intended to limit in any way the possible uses of the remote debugger functionality as disclosed herein, but merely represent some examples which those skilled in these arts will recognize as exemplary. The scope of the doeDebugger invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which the claims are entitled.

Related Application Filing Date (1):
19950303

CLAIMS:

1. A distributed debugger system for debugging a distributed target application system which resides on a local host computer and one or more remote host computers, the distributed debugger system comprising:

a debugger-GUI and one or more debugger engines (hereinafter termed "dbx engines"), said debugger-GUI providing an interface mechanism for communicating with said dbx engines, and for communicating with a user of said debugger system, wherein said dbx engines may reside on said local and remote host computers;

a communications mechanism for use by said dbx engines and said debugger-GUI in sending messages to and receiving messages from each other; and

a remote dbx engine which is one of said one or more dbx engines and which is residing on a host computer remote from said local host computer, said remote dbx engine connected to said debugger-GUI by means of said communication mechanism, said remote dbx engine co-operating with said debugger-GUI but capable of ignoring any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but are not part of a primary functionality of said target application system, permitting said user to debug said distributed target application system without being presented any intermediate IDL generated code mechanisms which connect local and remote sections of the target application system and thereby providing an illusion that the user is debugging a single process application.

2. A distributed debugger system for debugging a distributed target application system which resides on a local host computer and one or more remote host computers, the distributed debugger system comprising:

a debugger-GUI and one or more debugger engines (hereinafter termed "dbx engines"), said debugger-GUI providing an interface mechanism for communicating with said dbx engines, wherein said dbx engines may reside on said local and remote host computers, and for communicating with a user of said debugger system;

a communications mechanism for use by said dbx engines and said debugger-GUI in sending messages to and receiving messages from each other; and

a dbx Wrapper-Factory mechanism for use by said debugger-GUI to create a new dbx engine in a remote host computer for use in debugging a part of said target application system which resides on said remote host computer, said dbx Wrapper-Factory mechanism operable to generate said new dbx engine both at the initialization and during the course of execution of said target application system,

whereby the creation of said new dbx engine can be postponed until said new dbx engine is immediately needed by debugger-GUI.

3. The distributed debugger system for debugging a distributed target application system of claim 2 wherein said new dbx engine is connected to said debugger-GUI by means of said communications mechanism.

4. The distributed debugger system for debugging a distributed target application system of claim 2 wherein said new dbx engine in said remote host computer co-operates with said debugger-GUI and said one or more dbx engines residing on said local host computer to debug said target application system while ignoring any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself, thereby permitting said user to debug said distributed target application system with an illusion that the user is debugging a single process application.

5. The distributed debugger system for debugging a distributed target application system of claim 4 further comprising a dstep mechanism for instructing said dbx engines to ignore (that is, "step over") any of said intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself.

6. The distributed debugger system for debugging a distributed target application system of claim 2 further comprising a first communications mechanism to permit one of said dbx engines to communicate with another of said dbx engines regardless of whether these dbx engines are on different host computers.

7. The distributed debugger system for debugging a distributed target application system of claim 2 further comprising a second communications mechanism to permit said debugger-GUI to focus on one of said dbx engines regardless of which host computer said dbx engine is on.

8. The distributed debugger system for debugging a distributed target application system of claim 2 further comprising a third communications mechanism to permit said user to obtain from said debugger-GUI a list of all active dbx engines regardless of which host computer said dbx engines are on.

9. A dbx engine for use in a distributed debugger system for debugging a distributed target application system, said distributed target application system residing on a local host computer and one or more remote host computers, the dbx engine comprising:

a dstep mechanism for ignoring (that is, "stepping over") any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said distributed target application system but which are not part of said distributed target application system itself; and

a remote surrogate code mechanism for determining which intermediate IDL generated code mechanisms connecting local and remote sections of the distributed target application system should be ignored.

10. The dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 9 further comprising a remote breakpoint setting mechanism thereby permitting a user on a local host computer to set a breakpoint in a function of the distributed target application system which is actually implemented in a remote host computer.

11. The dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 9 further comprising a GetImplementation mechanism for locating a host ID and process ID (PID) of a server for any designated object.

12. The dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 9 further comprising an IdentifyRemoteFunction mechanism for identifying whether a remote dbx engine is running and if not for creating and attaching a dbx engine to a remote target function by using the facilities of a dbx WrapperFactory object.

13. The dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 9 further comprising a multiple dbx engine synchronizer mechanism for permitting dbx engines to communicate with each other.

14. A dbx engine for use in a distributed debugger system for debugging a distributed target application system, said distributed target application system residing on a local host computer and one or more remote host computers, the dbx engine comprising:

a dstep mechanism for ignoring (that is, "stepping over") any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself;

a remote surrogate code test mechanism for determining which intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system should be ignored:

a remote breakpoint setting mechanism thereby permitting a user on a local host computer to set a breakpoint in a function of the distributed target application system which is actually implemented in a remote host computer;

a GetImplementation mechanism for locating a host ID and process ID (PID) of a server for any designated object;

an IdentifyRemoteFunction mechanism for identifying whether a remote dbx engine is running and if not for creating and attaching a dbx engine to a remote target function by using the facilities of a dbx WrapperFactory object; and

a multiple dbx engine synchronizer mechanism for permitting dbx engines to communicate with each other.

15. A computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system, said distributed target application system residing on a local host computer and one or more remote host computers, the computer implemented method comprising the steps of:

providing said standard dbx engine which has capabilities equivalent to those of a SPARCworks dbx engine;

under computer control providing to said standard dbx engine a dstep mechanism for ignoring (that is, "stepping over") any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself; and

adding to said dbx engine, at any point during the course of execution of said target application system an IdentifyRemoteFunction mechanism for identifying whether a remote dbx engine is running and, if not for creating and attaching a dbx engine to a remote target function by using the facilities of a dbx WrapperFactory object.

16. The computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 15 further comprising the additional step of adding to said dbx engine under computer control a remote surrogate code test mechanism for determining which intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system should be ignored.

17. The computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 15 further comprising the additional step of adding to said dbx engine under computer control a remote breakpoint setting mechanism thereby permitting a user on a local host computer to set a breakpoint in a function of the distributed target application system which is actually implemented in a remote host computer.

18. The computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 15 further comprising the additional step of adding to said dbx engine under computer control a GetImplementation mechanism for locating a host ID and process ID (PID) of a server for any designated object.

19. The computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system of claim 15 further comprising the additional step of adding to said dbx engine under computer control a multiple dbx engine synchronizer mechanism for permitting dbx engines to communicate with each other.

20. A computer implemented method for modifying a standard dbx engine for use in a distributed debugger system for debugging a distributed target application system, said distributed target application system residing on a local host computer and one or more remote host computers, the computer implemented method comprising the steps of:

providing said standard dbx engine which has capabilities equivalent to those of a SPARCworks dbx engine; and

under computer control providing to said standard dbx engine a dstep mechanism for ignoring (that is, "stepping over") any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself;

providing to said standard dbx engine a remote surrogate code test mechanism for determining which intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system should be ignored;

providing to said standard dbx engine a remote breakpoint setting mechanism thereby permitting a user on a local host computer to set a breakpoint in a function of the distributed target application system which is actually implemented in a remote host computer;

providing to said standard dbx engine a GetImplementation mechanism for locating a host ID and process ID (PID) of a server for any designated object;

providing to said standard dbx engine an IdentifyRemoteFunction mechanism for identifying whether a remote dbx engine is running and if not for creating and attaching a dbx engine to a remote target function by using the facilities of a dbx WrapperFactory object; and

providing to said standard dbx engine a multiple dbx engine synchronizer mechanism for permitting dbx engines to communicate with each other.

21. A computer implemented method for producing a distributed debugger system for

debugging a distributed target application system which target application system resides on a local host computer and one or more remote host computers, the computer implemented method comprising the steps of:

providing in a local host computer a debugger-GUI and one or more debugger engines (hereinafter termed "dbx engines"), said debugger-GUI providing an interface mechanism for communicating with said dbx engines, and for communicating with a user of said debugger system;

providing a communication mechanism for use by said dbx engines and said debugger-GUI in sending messages to and receiving messages from each other; and

providing a remote dbx engine which is one of said one or more dbx engines and which is residing on a host computer remote from said local host computer, said remote dbx engine connected to said debugger-GUI by means of said communication mechanism, said remote dbx engine being able, under direction from said debugger-GUI, to attach itself to a section of said distributed target application system which is residing on said remote host computer for purposes of debugging said section of said distributed target application system which is residing on said remote host computer, said remote dbx engine co-operating with said debugger-GUI while ignoring any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself, thereby permitting said user to debug said distributed target application system with an illusion that the user is debugging a single process application.

22. A computer implemented method for producing a distributed debugger system for debugging a distributed target application system which resides on a local host computer and one or more remote host computers, the computer implemented method comprising the steps of:

providing a debugger-GUI and one or more debugger engines (hereinafter termed "dbx engines"), said debugger-GUI providing an interface mechanism for communicating with said dbx engines, wherein said dbx engines may reside on said local and remote host computers, and for communicating with a user of said debugger system;

providing a communications mechanism for use by said dbx engines and said debugger-GUI in sending messages to and receiving messages from each other; and

providing a dbx Wrapper-Factory mechanism for use by said debugger-GUI to create a new dbx engine in a remote host computer for use in debugging a part of said target application system which resides on said remote host computer, said dbx Wrapper-Factory mechanism operable to generate said new dbx engine both at the initialization and during the course of execution of said target application system,

whereby the creation of said new dbx engine can be postponed until said new dbx engine is immediately needed by debugger-GUI.

23. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 wherein said new dbx engine is connected to said debugger-GUI by means of said communications mechanism.

24. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 wherein said new dbx engine in said remote host computer co-operates with said debugger-GUI and said one or more dbx engines residing on said local host computer to debug said target application system while ignoring any intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself, thereby permitting said user to debug said distributed target application system with an illusion that the user is debugging a single process application.

25. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 further comprising a step of providing a dstep mechanism for instructing said dbx engines to ignore (that is, "step over") any of said intermediate Interface Definition Language ("IDL") generated code mechanisms which connect local and remote sections of said target application system but which are not part of said target application system itself.

26. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 further comprising a step of providing a first communications mechanism to permit one of said dbx engines to communicate with another of said dbx engines regardless of whether these dbx engines are on different host computers.

27. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 further comprising a step of providing a second communications mechanism to permit said debugger-GUI to focus on one of said dbx engines regardless of which host computer said dbx engine is on.

28. The computer implemented method for producing a distributed debugger system for debugging a distributed target application system of claim 22 further comprising a step of providing a third communications mechanism to permit said user to obtain from said debugger-GUI a list of all active dbx engines regardless of which host computer said dbx engines are on.

29. A computer-implemented method in a distributed computing environment of creating processes and communicating with said processes in said distributed computer environment, comprising the following steps:

a. a client process determining if services which are requested are remote;

b. if said client process determines said services are remote, then determining a remote location at which said services are located;

c. said client process sending a message to a server process having said services at said remote location to instruct said server to establish communications with said client process;

d. said server process spawning [a] an interface process for communicating with said client process;

e. said server process suspending operation until said interface process has attached to said server process;

f. said interface process attempting to attach to said server process;

g. if said interface process cannot attach to said server process, then said interface process alerting said client process, otherwise, said client process attaching to said server process and alerting said client process that said attachment has been successful;

h. said client process communicating with said server process via said interface process

i. said interface process receiving a quit command; and

j. said interface process responding to said quit command by detaching from said server process and subsequently deleting itself, execution of said server process being maintained subsequent to said interface process detaching and deleting itself.

WEST

Generate Collection Print

L3: Entry 8 of 54 File: USPT Sep 2, 2003

DOCUMENT-IDENTIFIER: US 6615253 B1
TITLE: Efficient server side data retrieval for execution of client side applications

Application Filing Date (1):
19990831

Detailed Description Text (57):
It includes components such as: Design/documentation tools Information repository Project Management tools Program Shells GUI Window painter Prototyping tools Programmer APIs Testing tools Source code control/build process Performance test tools Productivity tools Design tools Compiler/debugger Editor

Detailed Description Text (251):
If the design tool is used for programming, there are several features of a tool which must be considered. These features can have an impact on the productivity of programmers, performance of the applications, skill sets required, and other tools required for development. These features include: What programming language is supported? Is the programming language interpretive or compiled? Is it object oriented or structured procedural language? Does the tool support programming extensions to Dynamic Link Libraries? What are the debugging capabilities of the tool?

Detailed Description Text (361):
Database Services are responsible for providing access to a local or a remote database, maintaining integrity of the data within the database and supporting the ability to store data on either a single physical platform, or in some cases across multiple platforms. These services are typically provided by DBMS vendors and accessed via embedded or call-level SQL variants and supersets. Depending upon the underlying storage model, non-SQL access methods may be used instead.

Detailed Description Text (366):
Oracle 7.3; Sybase SQL Server; Informix; IBM DB/2; Microsoft SQL Server Oracle 7.3--market leader in the Unix client/server RDBMS market, Oracle is available for a wide variety of hardware platforms including MPP machines. Oracles market position and breadth of platform support has made it the RDBMS of choice for variety of financial, accounting, human resources, and manufacturing application software packages. Informix--second in RDBMS market share after Oracle, Informix is often selected for its ability to support both large centralized databases and distributed environments with a single RDBMS product. Sybase SQL Server--third in RDBMS market share, Sybase traditionally focused upon medium-sized databases and distributed environments; it has strong architecture support for database replication and distributed transaction processing across remote sites. IBM DB2--the leader in MVS mainframe database management, IBM DB2 family of relational database products are designed to offer open, industrial strength database management for decision support, transaction processing and line of business applications. The DB2 family now spans not only IBM platforms like personal computers, AS/400 systems, RISC System/6000 hardware and IBM mainframe computers, but also non-IBM machines such as Hewlett-Packard and Sun Microsystems. Microsoft SQL Server--the latest version of a high-performance client/server relational database management system. Building on version 6.0, SQL Server 6.5 introduces key new features such as transparent distributed transactions, simplified administration, OLE-based programming interfaces, improved support for industry standards and Internet integration.

Detailed Description Text (394):

Three key considerations are: Who owns and uses the data? Replication products support one or more of the three ownership models: Primary site ownership--data is owned by one site; Dynamic site ownership--data owned by one site, however site location can change; and Shared site ownership--data ownership is shared by multiple sites. Which of the, four basic types of replication style is appropriate? The four styles are: Data dissemination--portions of centrally maintained data are replicated to the appropriate remote sites; Data consolidation--data is replicated from local sites to a central site where all local site data is consolidated; Replication of logical partitions--replication of partitioned data; and Update anywhere--multiple remote sites can possible update same data at same time. What is the acceptable latency period (amount of time the primary and target data can be out of synch)? There are three basic replication styles depending on the amount of latency that is acceptable: Synchronous--real-time access for all sites (no latency); Asynchronous near real-time--short period of latency for target sites; Asynchronous batch/periodic--predetermined period of latency for all sites.

Detailed Description Text (429):
Products such as Lotus Notes and Microsoft Exchange allow remote users to replicate documents between a client machine and a central server, so that the users can work disconnected from the network. When reattached to the network, users perform an update that automatically exchanges information on new, modified and deleted documents.

Detailed Description Text (467):
Fax Services provide for the management of both in-bound and out-bound fax transmissions. If fax is used as a medium for communicating with customers or remote employees, in-bound fax services may be required for centrally receiving and electronically routing faxes to the intended recipient. Out-bound fax services can be as simple as supporting the sharing on the network of a single fax machine or group of machines for sending faxes.

Detailed Description Text (476):
FIG. 16 illustrates File Sharing services 1512. File Sharing services allow users to view, manage, read, and write files that may be located on a variety of platforms in a variety of locations. File Sharing services enable a unified view of independent file systems. This is represented in FIG. 16, which shows how a client can perceive remote files as being local.

Detailed Description Text (477):
File Sharing services can provide the following capabilities: Transparent access--access to remote files as if they were local Multi-user access--distribution and synchronization of files among multiple users, including file locking to manage access requests by multiple users File access control--use of Security services (user authentication and authorization) to manage file system security Multi-platform access--access to files located on various platforms (e.g., UNIX, NT, etc.) Integrated file directory--a logical directory structure that combines all accessible file directories, regardless of the physical directory structure Fault tolerance--use of primary and replica file servers to ensure high availability of file system Scalability--ability to integrate networks and distributed file systems of various sizes

Detailed Description Text (499):
Terminal services allow a client to connect to a non-local host via a network and to emulate the profile (e.g., the keyboard and screen characteristics) required by the host application. For example, when a workstation application logs on to a mainframe, the workstation functions as a dumb terminal. Terminal Services receive user input and send data streams back to the host processor. If connecting from a PC to another PC, the workstation might act as a remote control terminal (e.g., PCAnywhere).

Detailed Description Text (500):
The following are examples of Terminal services: Telnet--a simple and widely used terminal emulation protocol that is part of the TCP/IP communications protocol. Telnet operates establishing a TCP connection with the remotely located login server, minicomputer or mainframe. The client's keyboard strokes are sent to the remote machine while the remote machine sends back the characters displayed on the local terminal screen. 3270 emulation--emulation of the 3270 protocol that is used by IBM mainframe terminals. tn3270--a Telnet program that includes the 3270 protocol for logging onto IBM mainframes; part of the TCP/IP protocol suite. X Window

System--allows users to simultaneously access applications on one or more UNIX servers and display results in multiple windows on a local display. Recent enhancements to XWS include integration with the Web and optimization of network traffic (caching, compression, etc.). Remote control--While terminal emulation is typically used in host-based environments, remote control is a sophisticated type of client/server Terminal service. Remote control allows a client computer to control the processing on a remote desktop computer. The GUI on the client computer looks as if it is the GUI on the remote desktop. This makes it appear as if the remote applications are running on the client. rlogin--a remote terminal service implemented under BSD UNIX. The concept behind rlogin is that it supports "trusted" hosts. This is accomplished by having a set of machines that share common file access rights and logins. The user controls access by authorizing remote login based on a remote host and remote user name.

Detailed Description Text (504):
The following are examples of remote control products: Citrix's WinFrame Microcom's Carbon Copy Symantec's pcANYWHERE Stac's Reachout Traveling Software's LapLink

Detailed Description Text (584):
There are potential security issues associated with the execution of commands on a remote system. Some vendors install security features into their products. It is also possible for the architecture team to build additional security into the overall solution.

Detailed Description Text (588):
File Transfer services enable the sending and receiving of files or other large blocks of data between two resources. In addition to basic file transport, features for security, guaranteed delivery, sending and tracking sets of files, and error logging may be needed if a more robust file transfer architecture is required. The following are examples of File Transfer standards: File Transfer Protocol (FTP) allows users to upload and download files across the network. FTP also provides a mechanism to obtain filename, directory name, attributes and file size information. Remote file access protocols, such as Network File System (NFS) also use a block transfer method, but are optimized for online read/write paging of a file. HyperText Transfer Protocol (HTTP)--Within a Web-based environment, Web servers transfer HTML pages to clients using HTTP. HTTP can be thought of as a lightweight file transfer protocol optimized for transferring small files. HTTP reduces the inefficiencies of the FTP protocol. HTTP runs on top of TCP/IP and was developed specifically for the transmission of hypertext between client and server. The HTTP standard is changing rapidly. Secure Hypertext Transfer Protocol (S-HTTP)--a secure form of HTTP, mostly for fi nancial transactions on the Web. S-HTTP has gained a small level of acceptance among merchants selling products on the Internet as a way to conduct financial transactions (using credit card numbers, passing sensitive information) without the risk of unauthorized people intercepting this information. S-HTTP incorporates various cryptographic message formats such as DSA and RSA standards into both the Web client and the Web server. File Transfer and Access Management (FTAM)--The OSI (Open Systems Interconnection) standard for file transfer, file access, and file management across platforms.

Detailed Description Text (594):
The following are examples of File Transfer products: Computer Associates' CA-XCOM--provides data transport between mainframes, midrange, UNIX, and PC systems. XcelleNet's Remote-Ware--retrieves, appends, copies, sends, deletes, and renames files between remote users and enterprise systems. Hewlett-Packard's HP FTAM--provides file transfer, access, and management of files in OSI networks.

Detailed Description Text (597):
RPCs (Remote Procedure Calls) are a type of protocol by which an application sends a request to a remote system to execute a designated procedure using the supplied arguments and return the result. RPCs emulate the function call mechanisms found in procedural languages (e.g., the C language). This means that control is passed from the main logic of a program to the called function, with control returning to the main program once the called function completes its task. Because RPCs perform this mechanism across the network, they pass some element of control from one process to another, for example, from the client to the server. Since the client is dependent on the response from the server, it is normally blocked from performing any additional processing until a response is received. This type of synchronous data exchange is also referred to as blocking communications.

Detailed Description Text (602):
Message-Oriented Middleware is responsible for managing the interface to the underlying communications architecture via the communications protocol APIs and ensuring the delivery of the information to the remote process. This interface provide the following capabilities: Translating mnemonic or logical process names to operating system compatible format Opening a communications session and negotiating parameters for the session Translating data to the proper format Transferring data and control messages during the session Recovering any information if errors occur during transmission Passing results information and status to the application.

Detailed Description Text (642):
E-Mail takes on a greater significance in the modern organization. The E-Mail system, providing it has sufficient integrity and stability, can function as a key channel through which work objects move within, and between organizations in the form of messages and electronic forms. An E-Mail server stores and forwards E-Mail messages. Although some products like Lotus Notes use proprietary protocols, the following protocols used by E-Mail Services are based on open standards: X.400--The X.400 message handling system standard defines a platform independent standard for store-and-forward message transfers among mail servers. X.400 is often used as a backbone e-mail service, with gateways providing interconnection with end-user systems. SMTP--Simple Mail Transfer Protocol (SMTP) is a UNIX/Internet standard for transferring e-mail among servers. MIME--Multi-Purpose Internet Mail Extensions (MIME) is a protocol that enables Internet users to exchange multimedia e-mail messages. POP3--Post Office Protocol (POP) is used to distribute e-mail from an SMTP server to the actual recipient. IMAP4--Internet Message Access Protocol, Version 4 (IMAP4) allows a client to access and manipulate electronic mail messages on a server. IMAP4 permits manipulation of remote message folders, called "mailboxes", in a way that is functionally equivalent to local mailboxes. IMAP4 also provides the capability for an off-line client to re-synchronize with the server. IMAP4 includes standards for message handling features that allow users to download message header information and then decide which e-mail message contents to download.

Detailed Description Text (671):
Administration and systems management features such as remote management, remote configuration, backup and recovery, and disaster recovery should be considered.

Detailed Description Text (691):
Component Object Model (COM) is a client/server object-based model, developed by Microsoft, designed to allow software components and applications to interact with each other in a uniform and standard way. The COM standard is partly a specification and partly an implementation. The specification defines mechanisms for creation of objects and communication between objects. This part of the specification is paper-based and is not dependent on any particular language or operating system. Any language can be used as long as the standard is incorporated. The implementation part is the COM library which provides a number of services that support a mechanism which allows applications to connect to each other as software objects. COM is not a software layer through which all communications between objects occur. Instead, COM serves as a broker and name space keeper to connect a client and an object, but once that connection is established, the client and object communicate directly without having the overhead of passing through a central piece of API code. Originally conceived of as a compound document architecture, COM has been evolved to a full object request broker including recently added features for distributed object computing. DCOM (Distributed COM) contains features for extending the object model across the network using the DCE Remote Procedure Call (RPC) mechanism. In sum, COM defines how components should be built and how they should interact. DCOM defines how they should be distributed. Currently COM/DCOM is only supported on Windows-based machines. However, third-party vendors are in progress of porting this object model to other platforms such as Macintosh, UNIX, etc. FIG. 22 illustrates COM Messaging.

Detailed Description Text (718):
Design techniques for integration with existing systems can be grouped into two broad categories: Front end access--discussed as part of Terminal Emulation Back end access--tend to be used when existing data stores have information that is needed in the client/server environment but accessing the information through existing screens or functions is not feasible. Legacy Integration messaging services typically include remote data access through gateways. A database gateway provides an interface between the client/server environment and the legacy system. The gateway provides an ability to access and manipulate the data in the legacy system.

Detailed Description Text (748):
Filters Check Point FireWall-1--combines Internet, intranet and remote user access control with strong authentication, encryption and network address translation (NAT) services. The product is transparent to network users and supports multiple protocols. BorderWare Firewall--protects TCP/IP networks from unwanted external access as well as provides control of internal access to external services; supports packet filters and application-level proxies. Raptor System's Eagle Firewall Microsystem Software's Cyber Patrol Corporate Net Nanny Software's Net Nanny

Detailed Description Text (805):
The following are examples of products that perform Transport-level packet filtering: firewalls: Check Point FireWall-1--combines Internet, intranet and remote user access control with strong authentication, encryption and network address translation (NAT) services. The product is transparent to network users and supports multiple protocols. Secure Computing's BorderWare Firewall Server protects TCP/IP networks from unwanted external access as well as provides control of internal access to external services; supports packet filters and application-level proxies. Raptor Systems' Eagle Firewall routers: Cisco Systems Bay Networks 3Com Corp.

Detailed Description Text (900):
Cons of Using Tuxedo Tuxedo for basic c/s messaging is overkill. Expensive to purchase Can be complicated to develop with and administer System performance tuning requires an experienced Tuxedo administrator Uses IPC resources and therefore should not be on same machine w/other IPC products Must be understood thoroughly before design starts. If used incorrectly, can be very costly. Single threaded servers requires an upfront packaging design. Difficult to debug servers Does not work well with Pure Software products: Purify, Quantify Servers must be programmed to support client context data management Difficult to do asynch messaging in 3rd party Windows 3.x client tools (ex. PowerBuilder)

Detailed Description Text (933):
Profile Management Services are used to access and update local or remote system, user, or application profiles. User profiles, for example, can be used to store a variety of information such as the user's language and color preferences to basic job function information which may be used by Integrated Performance Support or Workflow Services.

Detailed Description Text (946):
Memory management, the allocating and freeing of system resources, is one of the more error prone development activities when using 3GL development tools. Creating architecture services for memory handling functions can reduce these hard to debug errors.

Detailed Description Text (1097):
18. Multiple Print Destinations: The report architecture should support distribution of reports for printing at centralized, remote, or local print sites without user or operations personnel intervention.

Detailed Description Text (1350):
Because components can be implemented in a variety of programming languages on a number of platforms, it is often necessary to have competencies in a number of technologies. For example, one client used Visual Basic, Smalltalk, C++, and COBOL for different layers of the system. The increasing number of technology combinations also increases the complexity associated with development activities such as testing, debugging, and configuration management.

Detailed Description Text (1434):
A component approach affects almost all aspects of the development lifecycle. For this reason the component learning curve cannot be equated with a programming learning curve such as `C`. There are multiple, distinct learning curves that affect individuals at many different levels in the organization: Component and object-oriented concepts and terminology Object analysis and design Programming language Programming environment and other development tools (e.g., browsers, debuggers, user interface tools) New architectures--such as how to use the project-specific application framework Management--such as estimating and planning for work, and managing iteration and prototyping

Detailed Description Text (1626):

Performance prototypes primarily address technology architecture questions. For example, the architecture team may need to decide early on whether to use messaging, remote procedure calls, or shipped SQL statements for distribution services between client and server. A prototype is often the only way to identify the most effective solution.

Detailed Description Text (1642):
The component must then be documented and rolled out for reuse to all developers. In many cases, the roll-out requires a formal group meeting to answer questions. During the support and refinement phase, the component is refined as other use cases generate new requirements, and bugs or performance problems are identified. Although the implementation details of the component should not be widely known, it is critical that developers thoroughly understand the purpose and public behaviors of the component. If they do not, then they may not be able to effectively reuse and debug interactions with it.

Detailed Description Text (1828):
As with a traditional client/server system, performance risks should be addressed early. Performance requirements often have a severe impact on the technology architecture including the infrastructure design and the platform systems software and hardware. For example, the architecture team may need to decide whether to use messaging, remote procedure calls, or shipped SQL statements for distribution services between client and server. Performance may also impact fundamental platform decisions such as the choice of language, DBMS vendor, operating system, network, or hardware configuration.

Detailed Description Text (1895):
In any complex information processing system, there will be a variety of different types of information, with a corresponding variety of actions which must be taken to process that information. One of the difficulties in this task involves taking an information source and creating an appropriate internal representation for it. The typical approach to this problem takes the form of a large switch/case statement, where each case deals with one of the information types. The switch/case approach leads to components that are very difficult to maintain, extend, debug, etc. and also leads to a procedural programming style. This approach also makes it extremely difficult to properly manage dependencies so that the details depend on the framework and not vice-versa.

Detailed Description Text (2018):
The performance characteristics of remote components are very different from "in process" components. The cost of requesting and transmitting data between remote components is much higher and should be considered in a distributed solution. As a result, distributed solutions often call for communication patterns that improve upon the performance aspects most important to the system. The Structure Based Communication pattern addresses the "chattiness" associated with distributed applications. It helps reduce network load and increases system response time. The Paging Communication pattern addresses the common need to retrieve and display large lists of data. It shows how incremental fetching can be used to provide much better perceived responsiveness in GUI based applications.

Detailed Description Text (2019):
The cost of locating a remote service and establishing a connection to that service can also be a costly endeavor. The Refreshable Proxy Pool pattern describes a robust and efficient way to minimize this "lookup" activity.

Detailed Description Text (2048):
Conversely, a stream could be created by a non-object system (or another object-based system for that matter) and sent to one's object-based system. In this case, CustomerObject could use a "streamOff: aStream" method and instantiate a new instance of an aCusiomerObject and populate it with the appropriate attribute values. Eagle Architecture Framework: Uses Stream Based Communication in a number of ways. First of all, it uses it to embed tracing information in CORBA distributed requests. Second of all, it is used to replicate state, between fault-tolerant services. MCI: Invoice Development Workbench. This workbench helps MCI create error-free invoice definitions for the various Local Bells. Stream-based communication was used as part of an efficient, lightweight persistence mechanism. Java Serialization: This is a Java defined fixed format for streaming objects. Object Request Brokers (ORBs) that use CORBA, DCOM, or Java Remote Method Invocation (RMI)--ORBs that use one of these standards implement this pattern. They define an

Interface Definition Language (IDL) that is the format or contract of the stream and use stream-based communication as the communication medium.

Detailed Description Text (2076):
1a. "Bind" the interface name (Update Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the Interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

Detailed Description Text (2080):
4. The Naming Service returns the Remote Object Reference (network location) for the Browsing Interface. The Proxy now has all the information it needs to access an operation on the Browsing Interface.

Detailed Description Text (2103):
Proxy Pool--The Proxy Pool pattern helps balance the cost of instantiating Remote Proxies and retaining Proxy "freshness." The Proxy Pool pattern can be used to create a pool of Proxies to Globally Addressable Interfaces.

Detailed Description Text (2181):
1a. "Bind" the interface name (Customer Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the Interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

Detailed Description Text (2184):
4. The Naming Service returns the Remote Object Reference (network location) for the Customer Interface. The Proxy now has all the information it needs to access an operation on the Customer Interface.

Detailed Description Text (2211):
When transmitting data across a network between a client and server application, the middleware's "type system" does not always support null values. How can a remote service send or receive null values over a communications medium that does not support them?

Detailed Description Text (2282):
In distributed systems with many clients, it is important to establish connections with remote servers in an efficient manner. In a manner where clients evenly utilize the available servers. How can this be performed in a consistent manner for all clients?

Detailed Description Text (2287):
Therefore, use a Refreshable Proxy Pool mechanism that standardizes the usage, allocation and replenishment of proxies in a client's pool. Initially, the Proxy Pool will allocate a bunch of Proxies to remote services using some sort of a Lookup Service (e.g. Trader Service, Naming Service). The Proxy Pool will hold onto these Proxies and allocate them to Clients as they need them. When the client asks the Proxy Pool for a proxy, the pool will hand out a new Proxy.

Detailed Description Text (2308):
Single Use--As opposed to pooling connections to a remote server a client can request a new connection each time a GAI is needed. This would work best when a client infrequently needs GAIs.

Detailed Description Text (2390):
This is an excellent programming model that frees the developer to access local and remote objects in the same fashion. Unfortunately, it makes it easier for the application developer to forget about the physical realities of the network. A network call is always slower than a call within a single machine. Ignoring this reality may result in an unacceptably slow application.

Detailed Description Text (2398):
Collaborations 1. The client instantiates a Proxy (Customer Component Proxy) to the Customer Component. The Client then asks the Proxy for Customer Jimbo Jones. 2. The Customer Component Proxy forwards the request across the network to the Customer Component. 3. The Customer Component requests the information for Jimbo Jones from the database. 4. The Database returns the data associated with customer Jimbo Jones. 5. The Customer Server Component instantiates a customer object using the Jimbo

Jones data from the database. 6. The Customer Server Component returns a remote object reference to the "Jimbo Jones" object running on the Server. 7. The Client creates a proxy to the "Jimbo Jones" object using the remote object reference.

Detailed Description Text (2495):
Benefits Development. Depending on the distribution model in place, business processing can be executed and tested before the appropriate views have been implemented. Automated testing. The View Configurer is particularly useful when you want to use scripts and avoid bringing up windows with automated testing. This is especially true for performance testing, where you might want to run 100 transactions, which might involve instantiating 100 instances of the same activity. Running processes in batch mode. The View Configurer allows processes to run without a View, and makes it very simple to connect, disconnect, or reconnect related views. Distribution Transparency. In a distributed environment, the process might live on a different machine from the end user's machine. In that case, it cannot launch the view directly, within its own executable. (Unless using a remote windowing system like X-windows, etc.) So the View Configurer allows application architects to transparently move process logic around, depending on the distribution model.

Detailed Description Text (2516):
Some condition checks may be expensive to complete (database and remote component queries). How can these be turned on and off to meet performance expectations? Problem with deferred evaluation; see below.

Detailed Description Text (2565):
The first step is to create an exception interface that all other interfaces will use or extend. It is not possible to provide one here as it greatly depends on the requirements at hand. But here are some guidelines: Determine the exception naming conventions. Use either a prefix or suffix to indicate that the interface is an exception. Also consider naming exceptions with the layer or domain they originate from. For example you may have an exception, CaAddressExcp, which is owned by the Customer Acquisition domain. Provide a means to determine where the error occurred (file, line, client or server, layer, . . .) so that it can be investigated. Provide a means to determine what happened (could not open file: XYZ). Provide context as to what was happening (Saving account information). Provide a way to stream the exception or stringify it. Consider separate production messages versus debug messages. Don't try to indicate severity. This is determined by the context of the caller, not the callee.

Detailed Description Text (2816):
Benefits Development Time. Data access and business logic can be developed in parallel reducing overall development time. Separation of concern. Data access remains separate from business logic, improving the understandability of the design. Testability. Business objects can be more easily developed and tested based on data access stubs, thereby relieving the business object development teams from dependencies on the data access classes and the database libraries. Caching and identity management. Separating the persistent state from the persistent object can be leveraged to aid in managing multiple class instances that represent the same entity (see the Object Identity Cache pattern). Object distribution. Separating the persistent state from the persistent object can aid in passing state in a distributed system. In cases where in is necessary to pass an object as an argument to a distributed method, it is more desirable from a performance perspective to pass the object's state as opposed to a remote reference. A new instance can then be created from the state, manipulated locally and then returned to the caller.

Detailed Description Text (2899):
Moreover, this is error-prone because it can be difficult to detect if the developer makes a mistake. For example, a new requirement could make the window display address information. In addition to re-painting the window, the developer would also need to modify their hand-coded methods. But the developer might forget to update the isDirty( ) or release( ) methods. Such errors can be difficult to locate. (Readers who have debugged memory leaks will certainly agree.)

Detailed Description Text (2997):
Although LUW contexts manipulate separate copies of business objects, they can often share the same retrieved data stream. For example, when a workstation retrieves data for Customer ABCD, the returned stream can be stored in a global cache. If another context wants to later instantiate its own copy of Customer ABCD, it can reuse the details stored in the stream cache. This improves performance, by avoiding a

redundant request to the remote data store.

Detailed Description Paragraph Table (21):
Class Description PooledProxy (10402) This is the base class for the pooled proxy. It actually acts as a wrapper for a Proxy and maintains all usage and reference counting information. ProxyPool (10404) This is the proxy pool, where clients go to retrieve a proxy. It should be thread-safe in that multiple threads are automatically synchronized. This pool should only contain valid proxies that have been allocated by the AllocationPool. When a proxy is requested, the usage count is incremented. After the "usage" passes retirement age, the proxy is remove from the pool and placed back into the allocation pool. AllocationPool (10406) This is the pool that actually does the proxy allocation. This pool is populated with unallocated proxies and a "reader" thread will allocate them. Since there will only be one, this class should be implemented as a Singleton. This pool however can allocate proxies of any type. ProxyHandle<T> (10408) This is the Handle that clients should use to manage pooled proxies. The handles must use a static_cast<T> (C++ template) to retrieve the correct proxy. T is defined by a client template instantiation, and assumes the client knows exactly what type of proxy the pool is actually holding. Clients must take care to assure that pools only contain proxies of one type.

Detailed Description Paragraph Table (28):
// Customer Component Code here public class CustomerComponent { // Put the data associated with a Customer Object // into a data Structure. This data structure // will be sent across the network to a client. public Customer getCustomer(String aCustomerName) { // Find the Customer in the database . . . // Instantiate the Customer Object Customer aCustomer = new Customer(.. . . . // Return a "remote object reference" to the // Jimbo Jones Customer object. return (aCustomer); } }

WEST

Generate Collection Print

L3: Entry 14 of 54 File: USPT Jul 8, 2003

DOCUMENT-IDENTIFIER: US 6591272 B1
TITLE: Method and apparatus to make and transmit objects from a database on a server computer to a client computer

Application Filing Date (1):
20000222

Detailed Description Text (18):
The result of the above technologies is a set of object-oriented networked remote database access methods that use generally accepted industry standard Internet-centric protocols and software engineering standards to extend persistent objects securely out from relational databases.

Detailed Description Text (48):
OSFORBStreams were originally designed to transmit blocks of objects via CORBA-2 using the GIOP mapping to TCP/IP (IIOP). Significant network traffic reductions also result using OSFORBStreams over the Pure Java Remote Method Invocation protocol known as RMI.

Detailed Description Text (87):
Use of CORBA-Standard IIOP Accessors NOTE: OMG IDL generated by OSF defines all attributes in all persistent relational CORBA objects as read-only. Thus the standard getter remote CORBA accessor methods are available but these do not use OSFORBStreams. This raw IIOP network traffic is not encrypted by default and thus the underlying TCP packets are open for viewing for anyone with a network protocol analyzer. The network packets are also more numerous as well.

Detailed Description Text (95):
A minor CORBA-2 limitation is that exceptions may only be thrown over the network when the synchronous, blocking remote accessors are used. Synchronous remote CORBA methods are available for all PRO-OBJECT server implementations, but no exceptions will be thrown over CORBA by default. This is because PRO-OBJECTS by default utilizes the asynchronous IIOP accessors, as identified by the Async suffix on the remote method call. Asynchronous IIOP remote method invocations are most commonly referred to as Distributed Callbacks and are the preferred way to invoke remote method calls via CORBA.

Detailed Description Text (96):
Thus a brief description of how ObjectServerFactory utilizes Distributed Callbacks is in order. A coherent mechanism to transport exceptions was needed which was independent of the type of remote method call used (asynchronous or synchronous). This is described in the next section.

Detailed Description Text (98):
OSF CORBA-style PRO-OBJECTS utilize by default asynchronous IIOP Distributed Callbacks. Early prototypes used standard synchronous method calls and this was not optimal. This is because the threads that issue the remote method call obviously block until the replies are received. If the thread that issued the remote IIOP method call was the AWT thread (a common occurrence), then the graphical interface would effectively be locked up until a reply was received from the server implementation.

Detailed Description Text (103):
Observe that one common callback object is used to reply to the requestor notwithstanding the type of remote operation (getobject( ), getobjectblock( ),

newobjecto( ) updateobject( ) or deleteObject( )). The type of reply is always an OSFORBStream and the event type sent in the initial request is copied to the reply in the in long/* ORBStreamEvent */event parameter of the ORBStreamReply callback object.

Detailed Description Text (106):
If an exception occurred on the remote operation, details regarding that exception will be placed in the in string exceptiondetail parameter (see IDL example in the previous section). If the remote operation completed normally and the reply OSFORBStream is valid, then exceptiondetail is zero-terminated, that is containing only the 0.times.00 byte.

Detailed Description Text (117):
Early on in the design of OSF it was decided to put each individual database I/O operation in a separate thread, so that one database request would not block or otherwise affect another request in any way. As a result the Persistence classes are multithreaded and by-design are guaranteed thread-safe. Since all types of object servers use the same persistence classes, all object servers are implicitly multithreaded.

Detailed Description Text (121):
The underlying database(s) is/are queried each time a persistent object method is invoked. This facilitates load balancing and recovery in addition to improving server performance, as there is no client or remote object-specific context data to store and fetch. Also, there are no upstream caches to synchronize when updates, deletes and inserts are made to the underlying database.

Detailed Description Text (128):
(10) Definition of a Remote Object Reference

Detailed Description Text (129):
Given the understanding of stateless object servers, we can now define an OSF-based remote object reference from the standpoint of the client or requesting middleware object. On OSF remote object is a combination handle and JavaBean state object to a set of persistent, relational server methods which are an object-oriented window to the underlying RDB.

Detailed Description Text (135):
Then only the attributes that are to be changed in the persistent relational object are added to the OSFORBStream. In addition to the attribute ID and the new attribute value, the old attribute value is added to the OSFORBStream as well. Given that PRO-OBJECTS can take the form of JavaBean components, it makes sense to handle the persistent relational update in the same manner as the update of a JavaBean bound property (in fact, that'is precisely what occurs: the attribute property is changed and then the remote RDB is synchronized, with the old, previous value of the attribute being sent to the server in the OSFORBStream). The OSFORBStream is then transmitted to the server implementation. A remote server exception will restore any changes made to bound properties and fire a PersistentObjectEvent.COMPONENTEXCEPTION to all registered event listeners.

Detailed Description Text (204):
It should now be apparent how ObjectServerFactory and the PRO-OBJECTS it creates to effectively and successfully address today's heretofore unsolved problems associated with building coherent, distributed persistent remote objects from relational database tables. Recall these problems involved distribution and communication mechanisms, performance, object lifecycle, locking, integration to legacy applications, recovery, scalability, and fault tolerance.

Detailed Description Text (214):
Object lifecycle issues are irrelevant for OSF-built persistent relational objects by-design. The reasons for this are: 1. OSF object servers do not use any caches upstream from the underlying RDB cache 2. Remote object server implementations are stateless and thus do not need to contain any client data 3. Any data in the client that is modified is first checked for currency on the server side, making sure the terminal operator is working with the most recent data when making changes. 4. An effective publish-and-subscribe model is employed to keep client data current when in changes in the underlying database

Detailed Description Text (240):

Thin Client architecture uses CORBA-2 object servers to distribute persistent, relational PRO-OBJECTS to clients or server middleware implementing domain or business logic. The Object Request Broker is contacted when a remote object is needed and the ORB via the IDL-generated stub code in the requester or client returns an object reference to the application.

Detailed Description Text (278):
The result of the above technologies is a set of object-oriented networked remote database access methods which uses generally accepted industry standard Internet-centric protocols and software engineering standards to extend persistent objects out from relational databases.

Detailed Description Text (306):
In addition to client-end and server-side persistent, relational object classes, OSF generates: OMG Interface Definition Language which exposes remote server methods to PRO-OBJECT based clients Build scripts for all generated code, including invocation of the IDL compiler and compiling IDL output A server registration script to register the CORBA server implementations with the Object Request Broker Master sedlanguage translation scripts to propagate translations to the various java.util.ListResourceBundle-derived objects HTML template files for data entry, inquiry and tabular display A Registry.java file containing all runtime parameters for a given installation, along with accessor classes and the object map Test programs for standalone testing of PRO-OBJECT component Other assorted utility and convenience scripts including a buildall script which builds everything in the proper sequence, interleaving builds into separate processes when possible

Detailed Description Text (308):
All CORBA distributed objects are derived from the BaseObject base IDL class, as outlined below. Note that this base class contains persistent relational distributed object accessors for reading objects, block object reads, object insertion, object update and object delete. Versions of these remote methods are supplied to the client PRO-OBJECT for both synchronous/blocking and asynchronous CORBA-based remote methods using familiar CORBA distributed callbacks. Asynchronous one-way persistent relational object methods use the ORBStreamReply callback object for all asynchronous remote object requests.

Detailed Description Text (311):
The pingObject( ) remote server method defined in the IDL excerpt above is used to synchronize initialization of the server by the PRO-OBJECT client component. This is because the _ bind( ) method to acquire a remote object reference has stopped blocking client code execution during server implementation in CORBA implementations. As a result, server methods could be invoked before the server implementation constructors completed. Thus methods which relied upon object references set by a constructor threw java.lang.NullPointerExceptions until the constructors completed.

Detailed Description Text (327):
In a product of this scale and complexity, we have found that the incremental time needed to initially use and enforce tight development standards pays off dramatically in faster debugging and easier longer term maintenance. Our software reads like a book, that is the idea.

Detailed Description Text (348):
Not that the significant advantages of this persistence model are: 1. Relational to Object and Object to Relational translation is independent from the network transport and runtime server environment or architecture 2. Persistence objects are inherently thread-safe by design 3. Blocking occurs only in one processing thread as a persistence processing thread is'started for each client request. Thus one client or object requester will not affect another. 4. When the thread containing the persistence object terminates, all data associated with the persistence object is immediately and explicitly garbage collected, satisfying numerous security audit requirements (because no client context data and security access parameters are left lurking about on the servers).

Detailed Description Text (547):
First, the client application has to request a dozen remote object references. In CORBA-land this is typically received as a sequence of object references. Network round trip count: 1.

Detailed Description Text (548):
Then with each object reference, one can dereference and use the object ref to access each attribute of the remote object. This requires an IIOP round-trip for each attribute in the object. There's eight attributes per object, thus network round trip count increases by eight per object or 96, plus the original sequence of refs is 97 or 194 network packets total to fill the 12*8 grid with data.

Detailed Description Text (558):
Although OSFORBStreams increase performance significantly, the IDL-defined accessor methods to the remote object are still available by the familiar getAttributeName( ) method calls.

Detailed Description Text (559):
Setter methods are not supported by individual attribute via the CORBA remote object reference because transaction synchronization over multiple IIOP network hops would leave records locked in the underlying RDBs for far too long of a time interval. The only solution which had reasonable performance and had nominal consequences was to set autocommit on a but this made rollback impossible when multiple attributes required update in a single transaction.

Detailed Description Text (560):
Thus, a far more coherent transaction bracketing scheme was implemented (refer to the OSFPersistenceObject base class in a previous section) and all remote object attributes in the OMG IDL are defined as read-only as a result.

Detailed Description Text (561):
Remote Persistent Object Requests with OSFORBStreams

Detailed Description Text (565):
Remote Persistent Object Reply Processing with OSFORBStreams

Detailed Description Text (675):
Use of ORBStreams in a distributed, remote object class is exactly the same as shown above in the Persistence class example. This is because the appropriate server implementation object (CORBA) or the javax.ejb.EntityBean-derived component server-side class uses the same Persistence interface as described above, providing a most practical, efficient and coherent separation between relational <-> object translation and the desired network transport scheme.

Detailed Description Text (677):
The client-end or requestor then makes the appropriate OSFORBStream request and injects the request into the underlying network transport. The client-end or requestor then blocks and waits (in the case of an synchronous persistent operation remote operation) or sets a reply event and carries on (asynchronous mode, very handy as it does not block import Java threads, such as the AWT thread).

Detailed Description Text (678):
The same examples as in the Persistence example above will now be demonstrated using OrbixWeb and the contents of each OSFORBStream `.vertline.`-delimited field described in detail. Be advised that this detail is hidden away from the developer since these remote persistent relational object method calls and event handlers are buried in the Java Bean component, which encapsulates the PRO-OBJECT. The developer only has to "wire" the events, methods and properties together using a JavaBean-aware IDE. However, if a developer desires to either understand the generated code exported by the IDE or desires to make a CORBA PRO-OBJECT request in server middleware (in a aserver-side only application with no GUI for example), demonstration of the Beans.instantiate( ) method and the underlying PRO-OBJECT access methods will prove informative and useful.

Detailed Description Paragraph Table (4):
```
/** * ##ObjectName##Object Java Component */ // Generated by ObjectServerFactory
(TM) // Copyright TriCoron Networks, Inc. 1998, Patent(s) Pending package
##Package##.client; import java.io.Serializable; import
java.awt.event.ActionListener; import java.awt.event.ActionEvent; import
java.beans.PropertyChangeSupport; import java.beans.PropertyChangeListener; import
org.omg.CORBA.ORB; import IE.Iona.OrbixWeb._CORBA; import
org.omg.CORBA.SystemException; import com.tricoron.OSFv13.OSFBaseObject; import
com.tricoron.OSFv13.OSFObject; import com.tricoron.OSFv13.OSFORBStream; import
com.tricoron.OSFv13.OSFORBStreamObject; import
com.tricoron.OSFv12.OSFSyetemManagement; import ##Package##.servercommon.Registry;
import com.tricoron.OSFv12.GeneralExceptionFormat; import
##Package##.servercommon.##ObjectName##Package.##ObjectName##AttributeIDs; import
##Package##.servercommon.ORBStreamEvent; import
##Package##.servercommon._ORBStreamReplyImplBase; import
##Package##.servercommon.GeneralException; import
##Package##.servercommon.MultiDBSynchronisationException; import
##Package##.servercommon.ORBStreamReply; import
##Package##.servercommon.##ObjectName##; import ##Package##.servercommon.ObjectID;
import ##Package##.servercommon.##ObjectName##Helper; public class
##ObjectName##Object extends OSFObject implements Serializable { // manifest
constants final int MAXKEYCOUNT = ##MAXKEYCOUNT##; final int ATTRIBUTECOUNT =
##ATTRIBUTECOUNT##; // includes all key fields // primary object instance attributes
##attributeblock## public String ##attributeName## = ""; ##endattributeblock## //
other instance vars public String beanname_ =""; public String keylist_ [ ] = new
String[MAXKEYCOUNT]; public String keyliststring_ = ""; // handy offsets for vectors
and matrices or attributes and descriptors ##attributeblock## public int
##ATTRIBUTENAME##ATTRIBUTEID = ##ObjectName##AttributeIDs._ ##ATTRIBUTENANE##AID;
##endattributeblock## // transient attributes protected static transient int
instancecount_ = 0; protected transient ORBStreamReply returnstream_ = null; //
##ObjectName##-specific transient attributes // remote proxy object declaration.
protected transient ##ObjectName## ##objectname##_ = null; // obligatory zero-arg
ctor public ##ObjectName##Object ( ) throws GeneralException { // create unique
component instance name synchronized (this) { instancecount_++; } beanname_ =
this.getClass( ).getName( ) + instancecount_; String hostname = locateServer( ); try
} // bind to IIOP proxy object ##objectname##_ = ##ObjectName##Helper.bind( ":" +
"ObjectName##Server", hostname); // establish asynchronous callback object
returnstream_ = new ##ObjectName##LocalImplementation(this); } <snip>
```

WEST

Generate Collection | Print

L4: Entry 15 of 20 File: USPT Sep 14, 1999

DOCUMENT-IDENTIFIER: US 5953530 A
**** See image for Certificate of Correction ****
TITLE: Method and apparatus for run-time memory access checking and memory leak detection of a multi-threaded program

Abstract Text (1):
The present invention is a system and method for a "debugger Run-Time-Checking for valid memory accesses for multi-threaded application programs" (hereinafter "RTC/MT") wherein a run-time process which includes multiple threads running either serially or concurrently, may be monitored by a debugger program and memory access errors detected and correctly attributed to the process thread encountering the error. The RTC/MT system of the present invention also provides an apparatus and method which monitors and reports memory leaks as required for multi-threaded target programs.

Application Filing Date (1):
19971125

Brief Summary Text (3):
This invention relates to the field of multi-processing computers, multi-threaded computer systems development and run-time debugging. More specifically, the invention is a method and apparatus for run-time memory access checking of a target multi-threaded system.

Brief Summary Text (5):
The invention described in this application is related to the debugger system described in U.S. Pat. No. 5,581,697 issued on Dec. 3, 1996, titled "Method and Apparatus for Run-Time Error Checking Using Dynamic Patching" by Wayne C. Gramlich, Sunnyvale, Calif.; Achut Reddy, San Jose, Calif.; and Shyam Desirazu, Foster City, Calif., and related to the system described in the U.S. Pat. No. 5,675,803 issued on Sep. 7, 1987 titled "Method & Apparatus for a Fast Debugger Fix & Continue Operation" by Thomas Preisler, Wayne C. Gramlich, Eduardo Pelegri-Llopart and Terrence Miller, both of which applications are hereby incorporated herein by reference.

Brief Summary Text (7):
Debugger programs written for uni-processor (i.e. single CPU) systems will generally not function correctly when testing application programs which are written to function in a multi-threaded mode. In the past, attempts have been made to develop debugging systems which check memory accesses during run-time but these debuggers are designed with uni-processor based application programs in mind. One such attempt was to interleave additional instructions adjacent to every memory access instruction in an object code module and then load and execute the augmented or new object code module in order to test the status of the addressed memory location during the execution of the augmented or new object code module. This method is used by the Purify program of Pure Software, Inc. which is described in U.S. Pat. Nos. 5,193,180 issued Mar. 9, 1993 and 5,335,344 issued Aug. 2, 1994. The Purify system reads object modules created by a compiler and interleaves instructions into the code of a target object module for every memory access instruction in the original object code module, thereby creating a new augmented object module which can then be linked to related object code and library modules and loaded into a computer and executed. This Purify approach is designed for single-threaded application programs and has been shown to incorrectly test a target application designed to be multi-threaded. This is due to the fact that each thread has its own Program Counter (PC) and stack and a debugger must be able to handle these separate stacks and

report errors according to the particular thread which contained the error. Sun Microsystems, Inc., the assignee of this invention, has a run-time-checking feature in its dbx debugger Run-Time-Checking (RTC) system which is sold under the title of SPARCWorks, a collection of several developer tools. Unlike the Purify product, Sun's debugger product operates on a target application by loading the original object code module into a computer under the control of the debugger and starting a process reflecting the target application. If run-time-checking is requested by the user, the RTC section of the debugger overlays every memory reference instruction with a branch to instrumentation code and library modules designed to test the validity of memory locations being accessed. However this RTC system itself was originally designed to operate on single-threaded processes and it too requires modification to handle concurrently operating multiple threads with their individual stacks and program counters, etc. It is desirable that run-time debugging and especially memory access checking tools be available for multi-threaded application programs.

Brief Summary Text (10):
The present invention overcomes the disadvantages of the above described systems by providing an economical, high performance, system and method for debugging a multi-threaded target program using a memory access checking system which is itself multi-thread safe. More specifically, according to one aspect of the invention, a computer implemented method for memory access checking of a multi-threaded target program is claimed, wherein a debugger program which does the checking is itself multi-thread safe ("MT safe") and wherein this MT safe debugger maintains a status of all memory locations as they are allocated and deallocated by the target program and thereafter reports any errors which may occur when the target program attempts to access a memory location in a way which is deemed invalid for that location.

Brief Summary Text (11):
According to a second aspect of the invention a computer system for memory access checking of a multi-threaded target program is claimed, wherein a multi-threaded operating system and a multi-thread safe debugger mechanism operate to maintain memory location status and to check this status and report any errors that occur when the target program accesses a location in an invalid way:

Brief Summary Text (12):
According to another aspect of the invention, a method and an computer system are claimed, wherein a multi-threaded safe debugger system maintains memory leak status and reports errors when required indicating any "memory leaks" which are defined as memory locations which were allocated but which are inaccessible by the target program. Such leaks occur either because a routine may terminate without freeing up previously allocated memory which is no longer used or because a pointer to the allocated memory somehow was destroyed or deleted so that the memory location is no longer accessible.

Drawing Description Text (8):
FIG. 6 illustrates in block diagram form the steps performed by the basic debugger to accommodate multi-processing.

Drawing Description Text (9):
FIG. 7 illustrates in block diagram form the steps performed by the Run-time-checker (RTC) and "librtc.so" portions of the basic debugger to accommodate multi-processing when doing memory access checking.

Drawing Description Text (10):
FIG. 8 illustrates in block diagram form the steps performed by the RTC and "librtc.so" module basic debugger to accommodate multi-processing when doing memory leak checking.

Detailed Description Text (7):
Apparatus and methods for dynamic patching for run-time checking and for rapid debugging of a multi-threaded target program are disclosed. In the following description, for purposes of explanation, specific instruction calls, modules, etc., are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well known circuits and devices are shown in block diagram form in order not to obscure the present invention unnecessarily. Similarly, in the preferred embodiment, use is made of uni-processor and multi-processor computer systems as well as the Solaris

operating system, all of which are made and sold by Sun Microsystems, Inc. however the present invention may be practiced on other computer hardware systems and using other compatible operating systems.

Detailed Description Text (8):
The present invention is a system and method for a "debugger Run-Time-Checking for valid memory accesses for multi-threaded application programs" (hereinafter "RTC/MT") wherein a run-time process which includes multiple threads running either serially or concurrently, may be monitored by a debugger program and memory access errors detected and correctly attributed to the process thread encountering the error. The invention described in this application is related to the Run Time Checking system described in U.S. patent application Ser. No. 08/189,089 filed on Jan. 28, 1994, titled "Method and Apparatus for Run-Time Error Checking Using Dynamic Patching" by Wayne C. Gramlich, Sunnyvale, Calif.; Achut Reddy, San Jose, Calif.; and Shyam Desirazu, Poster City, Calif., and related to the system described in the U.S. patent Continuation-in-part application Ser. No. 08/299,720 filed on Sep. 01, 1994 titled "Method & Apparatus for a Fast Debugger Fix & Continue Operation" by Thomas Preisler, Wayne C. Gramlich, Eduardo Pelegri-Llopart and Terrence Miller, both of which applications are hereby incorporated herein by reference. The first of the two applications identified above (the parent of the two applications) discloses and claims Run Time Checking related to a target application program being debugged while the second continuation-in-part application additionally discloses and claims the Fix and Continue error processing system for debugging the target application program. The parent application utilizes dynamic patching to check for program errors during program run-time which are not detected by the compiler. Such run-time errors are checked by patching a run-time process corresponding to the program such that at every point where the program is about to access memory, the program instead branches to a different location where checks on the memory address that is about to be accessed are performed. If the memory address that the program is about to access is invalid, an error is recorded, otherwise if the memory address is valid then the program will continue execution. The actual patching process is done inside the RTC module. It will be recognized that if the target program to be debugged is a multi-threaded program then the debugger not only must be able to keep track of whether multiple threads are executing concurrently, but must itself be capable of handling multiple accesses of its routines in a safe way. That is the RTC module must be multi-thread safe ("MT safe"). If the RTC module is testing a multi-threaded application program process then RTC must recognize that stacks of other threads exist and therefore accesses to memory locations on these other stacks are legal accesses, and therefore each check for an error must be made with knowledge of the activities of all threads and each error detected must be reported with reference to the particular thread wherein the error was observed. Such multi-threaded error checking capability is the subject of the present invention claimed in this application. In the sections which follow, the preferred embodiment is described as a modification of the Sun Microsystems, Inc. single-threaded system of Run-Time Checking, which is described in detail in the aforementioned parent application which is incorporated herein by reference and which for completeness, is described in some detail below. While the multi-threaded version of run time checking (RTC/MT) will operate on computer hardware with one CPU or multiple CPUs, it is clear that multi-threaded applications are most effectively run on multi-processor systems. Therefore this description is followed by a summary description of a typical multi-processor configuration capable of executing multi-threaded processes concurrently. It will be appreciated that the present invention may be easily adapted to function on any number of vendor's multi-processor systems such as IBM, Hewlett Packard, DEC, MIPS, etc. and to function with target application programs to be debugged from various software vendors such as IBM, Hewlett Packard, DEC, MIPS, Microsoft, Novell, etc.

Detailed Description Text (11):
FIG. 1 illustrated the single threaded RTC system. As shown in FIG. 1, a target program image is read into a debugger program 307 (dbx) through an I/O device 304, and stored in memory to provide an in-memory copy 308 of a program 302. A module within the debugger program 307 referred to as a "run-time checking" (RTC) module 309 handles the user interface, printing of error messages and also handles the patching of the in-memory process 308 corresponding to the program 302. A shared library (Libraries) module 310 is loaded into the computer memory 305, and performs the run-time checking. In the preferred embodiment the principal library routine used is designated "librtc.so".

Detailed Description Text (12):

This in-memory copy of the program (the process) 308 becomes a patched process, called "instrumented program" herein. The patches are applied only to this in-memory copy 308 of the target program and not to the original program 302 stored on disk 301. Therefore, the original file 302 never changes and there is never any relinking of the file required for the executable program. In addition, the program 302 does not have to be pre-patched. Instead, the patches are applied when the checking is initiated. The choice by the user is therefore delayed until the actual run-time rather than before execution. The CPU 306 controls the program execution of the debugger 307 and the program under test 308. The CPU 306 contains a Program Counter ("PC") 312 which points to the next instruction to be executed.

Detailed Description Text (13):
The Sun dbx debugger program 307 can dynamically load libraries at run-time that were not specified at link time. Since such loading of libraries is done dynamically in the debugger program 307, the RTC module 309 can trap all calls to load a new library in the program and may apply patches just before such libraries are executed.

Detailed Description Text (14):
In summary, with the Sun dbx debugger there is no necessity for pre-patching a program before execution. Instead, the patches may be applied when the checking is initiated, thereby delaying the choice of the user until the actual run-time. Furthermore, by not modifying the target program object code at all and thus eliminating the need to relink the object files to produce the executable program, the approach of the present method avoids the use of extra links. Finally, the patches are applied to an in-memory process initiated from the existing target program such that a fully instrumented process is achieved.

Detailed Description Text (15):
Reference is now made to FIG. 2, wherein a general flow chart for the method of dynamic patching for the run-time checking (hereinafter "RTC") in the Sun dbx debugger is illustrated. In order to detect memory access errors, all memory access instructions, including accesses to the stack and system calls that access user memory are intercepted. Such memory access instructions are then verified as to validity of memory address being accessed before continuing instruction execution.

Detailed Description Text (17):
As illustrated in FIG. 2, block 100, space is allocated for the patch tables and the patch tables and values are initialized. Next, as illustrated in block 110, the program to be error checked is initially read and loaded as it exists on the disk file. Such program is normally loaded in portions (load objects) as they are accessed by the user. However, by going through the steps illustrated in FIG. 2, the debugger will cause essentially all of the program to be accessed. Thus, as a result, when the debugger program has completed its processes, all of the program will have been patched. This debugger program is a special process that is able to read and write other processes and therefore able to change the program image process that is within the memory. All operations described within FIG. 2 are performed by the RTC module within the debugger program. As can be appreciated by FIG. 2, block 130, the debugger program creates a list of load objects. The load objects contain segments/functions within the program which have memory access instructions. The program may consist of a number of these load objects. The first type of load object is the program's main routine, which is the user part of the program. There are also shared libraries that the program uses, which are another type of load object. Both types of load objects are required in order to run the program. Once the debugger program has received a list of the load objects, it will scan the load objects, searching for instructions that it is going to patch later on. The only part of the load object the debugger program looks at during this instruction-by-instruction scan are the instructions themselves, i.e., the text, but not the data.

Detailed Description Text (18):
While the debugger program is identifying the patch sites, the debugger program also accumulates information regarding these patch sites, including patch site address, patch area address, patch type (i.e. memory access instruction type), whether a particular patch site should be patched, and the size of memory being accessed. Every load object has a table for the aforementioned patch site information, with one entry in the table for each patch site. The patch type or the type of memory access instruction for patching purposes defines the size of its corresponding section of patch area where the error checking is processed. A check command or

uncheck command issued by a user for a particular patch site will indicate whether or not errors will be reported for that particular patch site. More specifically, the check command indicates that the particular patch site should report errors and the uncheck command conversely indicates that errors for the particular patch site should not be reported. At the very end of the scan, the debugger program comes up with a total size of the section of patch area that the debugger program is going to need in order to accommodate the patch sites found. The identification of a patch site only needs to be done once for a load object and any subsequent execution pass only requires locating a space for the corresponding section of the patch area space and installing the patch. The total size needed for the patch area section is recorded and a list of the patch area section sizes is produced. This list of patch area section sizes is the input to the next step, step 140, in which memory space is actually allocated to the patch area. In step 140, the debugger program takes the list of patch area section sizes and attempts to allocate space for them. The debugger program first creates an initial map of the address space to see where everything is laid out. The system maps the load objects in various places in memory. Such mapping of load objects is not necessarily contiguous and there are holes in the address space. The job of the debugger program is to identify these holes and to map these lists of requests for space needed to these holes.

Detailed Description Text (19):
In one embodiment of the Sun dbx debugger RTC program, the address space data is accessed to obtain a list of all segments in the address space along with each segment's starting address and size. These segments may consist of text, data, stack and/or heap segments. The space between such segments, called "holes" herein, are used to allocate space for the sections of the patch area. A list containing the starting address of each text segment, ending address of each text segment and the size of sections of the patch area, sorted by ascending order of the starting address of each text segment, is obtained from the previous step 130. In step 140, a list of holes with starting addresses and segment sizes of the holes sorted by starting address of each hole is generated. The aforementioned holes are compared to the sizes of sections of the patch area needed by first checking for holes with address locations higher than the patch sites for the corresponding sections of the patch area. Given a hole with a size larger than the size of the section of the patch area for which space is to be allocated and the hole is not immediately before a stack segment, then the section of the patch area is allocated the hole space. After going through the list of the patch area section sizes and the list of the holes and allocating the holes to the sections of the patch area, the list of unallocated patch area sections produced will be scanned in descending order. The holes at lower addresses than the patch sites which are greater than or equal to the corresponding sections of the patch area are searched. The holes which are greater than or equal to particular section of the patch area are allocated to that section of the patch area. Such section of the patch area is placed at the bottom of the hole. Any patch sites for which its corresponding section of patch area is not allocated at the end of this step is not patched and the user will be warned that the request for error check was not met. In step 150, the system takes the information of where it found all the sections of the patch area and stores that information in the patch table and updates the address information in these patch tables.

Detailed Description Text (22):
Essentially, all the steps illustrated in FIG. 2 from steps 100 to 160 are performed when the user wishes to run the target program (i.e. execute the process) within the debugger program. In sum, steps 100 through 160 completes the patching for all the load objects that exist at the time the program is started.

Detailed Description Text (23):
In addition, the debugger program is able to load new load objects dynamically which were not in the program at the time the program was started. The system traps all calls to new load objects, and when the debugger program sees that a program is about to load a new object, the debugger program goes through a similar set of steps. The steps 110, 120, 200, 140, 150, 160 and 170 illustrate dynamic loading of a new load object. The steps are identical to the previously followed steps except there is no initialization. The global initialization is performed once only and then these steps are performed for each new load object that are dynamically loaded.

Detailed Description Text (24):
As illustrated in steps 175, 180 and 185 the debugger program is able to also

de-install the patches and dynamically unload load objects. When a de-install command is received steps 175, 180 and 185 are executed. At step 175, given a patched function, the page containing the patch site to be de-installed as well as the page containing the corresponding section of the patch area are read. The original instruction is then obtained from the section of the patch area and the branch to patch area instruction in the patch site is replaced by this original instruction. In addition to this replacement of the patch instruction in the patch site, user breakpoints at these patch sites will require replacing the patch instruction in the breakpoint data structure associated with the patch site as well. In the event that the patch site was not patched, a warning is issued to the user and nothing is de-installed. The user issuing the check command will merely replace the instruction at the patch site with the branch to patch area instruction.

Detailed Description Text (27):
FIG. 3 illustrates the dynamic patching for the run-time error checking method used in the Sun dbx debugger. A target program consists of a number of load objects and in turn the load objects contain a number of functions, and, function 10 as function foo, is one example. Such function will have a number of memory access-related instructions. One such instruction is illustrated as load instruction 40. The run-time checking (RTC) module will patch each and every such instruction for every load object that it is patching. This run-time checking (RTC) module scans each and every individual instruction that needs to be patched, as was illustrated by Box 130 of FIG. 2, and the original instructions are then replaced by unconditional branch instructions to the patch area. The location of the instruction that is patched is called "the patch site" 20. Therefore, if there was a load instruction at a location within a load object then that location would be called "a patch site" 20. The memory locations where the error checking is done is called "the patch area" 50. For each patch area 50, there will be one or more sections of the patch area 60, each section corresponding to a unique patch site. Therefore if there are 1,000 patch sites, there will be 1,000 sections of the patch area.

Detailed Description Text (28):
For each instruction that is replaced within the load object, there is an instruction to branch to the corresponding section of the patch area 60. Thus, there is a custom section of the patch area 60, in a given patch area 50 that is assigned to the whole load object for each patch site 20 and each patch site 20 is replaced with a branch to its own custom section in the patch area 60. These sections of the patch area 60 consist of several instructions which are basically set up to call some real checking codes 70 in a separate area in memory. In the preferred embodiment, this real checking code 70 is designated the library routine "librtc.so". Thus, "librtc.so" is called from the patch area 50 which performs the checks. If there are any errors to report, "librtc.so" will record the error in an error buffer from which the debugger program will report them, otherwise the process is returned to the patch area 60 and in turn the process is returned to the next instruction that will be executed in the user program. There are different types of sections of the patch area depending upon the types of instruction being patched. There are also several different kinds of cases due to delayed branch instructions that have to be handled separately. Therefore sections of the patch area 60 are not identical and the "librtc.so" routine may make different kinds of tests depending on the different ways in which it is called by the instrumenting instructions in the section of the patch area 60. In summary, a section of the patch area is specifically for one particular patch. FIG. 3 illustrates a process in which patch sites are replaced by branches to a section of the patch area 60 and a further branch to a checking code 70 and back to the next instruction to be executed in the user program. There are other cases that may modify the illustration in FIG. 3. For example, if an instruction to be patched was in a delay slot of a branch, i.e., a delayed branch instruction, then after branching to the patch area and the checking code, the process should branch to the address location the process was supposed to branch to prior to the error checking instead of branching back to the next instruction in sequence. In order to handle a target application process written for multi-threading (MT), several of these patch areas and error check routings must be modified as will be described in more detail below. To understand these modifications it is first necessary to describe a typical multi-processing environment.

Detailed Description Text (30):
FIG. 4 depicts a representative multi-processor machine configuration which would be typical for use with a multi-threaded target program. It should be noted however that multi-threaded programs can run on single-processor systems as well as

multi-processor systems but they just do not run as efficiently on a single-processor system. The present invention, RTC/MT can run on either type of system. In the preferred embodiment SunOS 5.0 is the operating system used which is part of the Sun Solaris Operating Environment. SunOS 5.0 is intended to run on tightly-coupled shared memory multi-processor systems with one or more processors. Referring now to FIG. 4, the typical multi-processor computer system is assumed to have one or more central processor units (CPUs)410,412,414 sharing a memory 420 and clock 418. The operating system kernel 416 assumes all processors are equivalent. Processors 410, 412, 414 execute kernel threads selected from the queue of runnable kernel threads 426. If a particular multiprocessor implementation places an asymmetric load on the processors (e.g., interrupts) the kernel 416 will nonetheless schedule threads to processors 410,412,414 as if they were equivalent. In general, all processors 410,412,414 see the same data in memory 420. This model is relaxed, somewhat, in that memory operations issued by a processor 410,412,414 may be delayed or reordered when viewed by other processors. In this environment, shared access to memory is preferably protected by synchronization objects 424. (The data locking mechanisms are also sometimes called synchronization variables or synchronization primitives). The exception is that single, primitive data items may be read or updated atomically (e.g. all the bytes in a word change at the same time). (A "word" is a four byte piece of data.) The shared memory 420 is assumed to be symmetrical. Thus the kernel 416 currently does not ensure that processes scheduled on a particular processor 410 (for example), are placed in a particular piece of memory 420 that is faster to access from that processor 410. It is possible for a kernel 416 to run "symmetrically" on a multiprocessor yet not allow more than one processor 410,412,414 to execute kernel code 416. This is clearly not a strategy that scales well with increasing numbers of processors, and in the preferred embodiment of the present invention, all of the processors 410,412,414 in the system can execute the shared kernel code 416 simultaneously, and use the data structures in the shared memory 420 to communicate between the processors 410, 412, 414 as required. Accordingly, when debugging a process that may have multiple threads concurrently accessing the same memory location it is essential for the debugger to be able to tell whether the memory location is allocated to some thread other than the thread which accessed it. That is, the memory location being accessed by thread 1 may be on the stack of thread 2 and if so is a valid memory location. The prior art debuggers would report this latter case as a memory access error incorrectly.

Detailed Description Text (48):
As indicated above, the preferred embodiment of the present invention makes use of the Sun Solaris Operating System, the Sun SPARCWorks debugger ("dbx" debugger) which includes the run-time-checking (RTC) routine, which itself makes use of the generalized memory status maintenance and memory status checker capabilities of the library routine "librtc.so". A target application program is loaded into a machine for testing under the control of the debugger and when run-time-checking is specified by the user, the RTC section of the debugger patches the target application program process and the "librtc.so" library routine is used in various ways and modes by each type of memory access patch code to maintain memory status and to check memory access. In order to modify this system to handle multi-threaded target application program processes, it was necessary to make the following general modifications to the uni-processor debugger/RTC system:

Detailed Description Text (51):
SPARCWorks debugger ("dbx")

Detailed Description Text (74):
Referring now to FIG. 6 the steps performed by the basic debugger ("dbx") are depicted 600. On beginning a debugger test run the target application object code is loaded into a machine under the control of dbx 602. The user selects a test mode 604 indicating whether he/she wants to do memory access checking only 605 or memory leak detection only 609 or both 607. Whatever the selection the dbx sets a mode indicator 606, 608, 610 and continues. The dbx then determines whether the target application is a multi-threaded application or not 612. If the target application is not a multi-threaded application 614 the dbx sets a single-threaded (or non-MT) indicator 618 and continues 630 calling the RTC section for further processing. If the target application is a multi-threaded application 616 dbx loads the additional multi-threaded library "libthread.sub.-- db 620 if it is not already loaded. The dbx sets the multi-threaded test indicator and continues 630 calling the RTC section for further processing.

Detailed Description Text (76):

If the entry to RTC is a memory access check 708 then RTC tests to see if memory accesses are to be tested or skipped 728 (the user can designate locations to be tested or not). If access checking is to be skipped then RTC is exited 714. If access checking is not to be skipped then RTC again tests whether it is a multi-threaded application 730. If it is not 734 then librtc.so does the normal (no threads) memory status test 746, records any errors if any 752 and exits 714. If the multi-threaded indicator is on 732 then the current thread ID is obtained from "libthread" 736, and similar to the above, if it is the first encounter by RTC with this thread 737, then RTC allocates storage for the per-thread data which includes the thread's ID, stack start address, stack limit, stack size, current stack pointer, a flag to indicate whether RTC is ON/OFF at that moment for that thread, error message buffer, and a flag to indicate whether the thread has been seen by RTC before 739. This per-thread data is maintained in a table as indicated above. The code in the RTC is then locked 740 so that an uninterrupted status check 742 may be made for the indicated location and then the checking code in RTC is unlocked 744. After the memory location's status is checked, the status is assessed for validity 746 and if valid 748 RTC is exited 714. If the location was found to be invalid 750 then an error message is recorded in the error buffer for the thread in question 752 recording the thread ID and the error type and location. Thereafter the RTC is exited 714. Note that the recorded error messages are typically displayed at the end of the debugging run or they can be displayed to the user as they are encountered. The user may specify which option he prefers by interacting with the debugger interface screen.

Detailed Description Text (77):
The dbx debugger and its RTC section has the capability of maintaining status for memory locations in order to detect "memory leaks." A "memory leak" is defined as a memory location which was allocated at some time (by creating a pointer to the location for example) but which no longer is capable of being accessed and yet the location has not been freed (i.e. unallocated; made available for further use.). This could happen for example by the pointer to the location getting changed without freeing the original location, or the routine containing the pointer simply being exited without freeing the location. Keeping track of such happenings in order to inform the user/developer of such inaccessible locations is the function of the "memory leak detection" feature of RTC. The user can specify that he/she wants all leaks displayed at the end of the debug run or at anytime he can specify "show leaks." Referring now to FIG. 8, the functions of RTC to handle memory leak detection in a multi-threaded environment 800 are depicted. When RTC is entered 630 it checks the entry type 802 and determines whether the entry is to report all leaks 804, report leaks now 810, change memory leak status 808 or initialize the memory leak status area 806. If it is an initialization entry 806 the memory area used by RTC for keeping track of leaks is initialized 812 and the program exits 816. If the entry is to update the memory leak status area 808 then the status is updated 818 and the program exits 816. The other two entries, report all leaks 804 and report leaks now 810 function the same way the only difference is the former occurs at the end of the debug run and the latter can occur at any time. Both entries go to check whether any threads are still alive 820. Typically at the end of the debug run all threads should be completed. If no threads are alive 822 then the leak memory status area is checked and all designated leaks are reported 826 and the program exits 816. Then dbx uses libthread-db to determine if there are any threads still active 824, (since libthread.sub.-- db provides a function for listing all active threads and since dbx maintains this list of all threads that have been created by the user process), and then RTC gets the next live thread ID from this list of active threads 828 and using that thread ID gets that thread's register set, thread stack size, and stack start address from libthread.sub.-- db and checks these to see if they contain any pointers to previously allocated memory and if so then the leak memory status area is updated to make locations corresponding to any found pointers be designated as "no leak." 830. The RTC program then checks to see if there are any more remaining live threads 832 and if so steps 828 and 830 are repeated. If all live threads have been checked 836 then the memory leak status area is checked and all leaks reported 826 and the program exits 816.

Detailed Description Text (78):
The preferred embodiment of the run-time-checking system for multi-threaded programs (RTC/MT) has been described in terms of specific procedures, structures (such as a typical multiprocessing hardware configuration), tests and in the framework of the Sun SPARCWorks debugger with a specific implementation of the Sun run-time-checking (RTC) feature and using specific Sun library routines such as "libthread" and "libthread.sub.-- db". However, those skilled in these arts will recognize that all

of these functions may be realized on various kinds of uni- or multi-processing hardware systems with various Operating Systems capable of executing multi-threaded applications. Similarly other equivalent testing systems may not use libraries such as "libthread" and "libthread.sub.-- db" to get the current thread ID and thereafter get that thread's stack, error buffer and register set and instead may use other devices for perceiving the existence of threads such as testing the thread stack size to see if it has grown beyond an expected size. Such apparently large stack sizes can be used as an indicator of a new stack and therefore a new thread. All such equivalent schemes are deemed to be equivalent to the preferred embodiment disclosed herein and claimed as follows.

Related Application Filing Date (1):
19950207

CLAIMS:

1. A method for memory access checking of a multi-threaded target program, said method executable on a computer system having a memory, a clock, one or more central processing units (CPUs), an I/O device for receiving inputs, an I/O device for sending outputs to, and at least one peripheral device, said computer system having program machine instructions in said memory, said computer system also having a multi--threaded operating system, said method comprising the steps of:

providing a multi--threaded safe ("MT safe") debugger program having memory access checking facilities, where being "multi-threaded safe" means that the debugger program itself is capable of handling multiple accesses of its routines in a safe way, and wherein the MT safe debugger program can operate in conjunction with said multi-threaded operating system;

providing memory status information for memory locations in said memory, said memory status information indicating at least whether a memory location is in an allocated state or in an unallocated state, wherein said allocated state corresponds to a memory location

allocated by a computer program and said unallocated state corresponds to a memory location not allocated by said computer program, said status information being maintained by said MT safe debugger program; and

under the control of said computer system, said MT safe debugger program checking said memory status information for each memory location accessed by said multi-threaded target program.

2. The method of claim 1 wherein said MT safe debugger program maintains said memory status information in a multi-threaded safe manner by using synchronization primitives to lock-out concurrent accesses to it until said memory status is updated or checked for a current thread, after which time said accesses are unlocked.

6. The method of claim 1 comprising the additional steps of:

providing memory leak status information for memory locations in said memory, said memory leak status information indicating at least whether a memory location is in an inaccessible state or not, wherein said inaccessible state corresponds to a memory location which is in an allocated state but which is inaccessible in said multi-threaded target computer program, said memory leak status information being maintained by said MT safe debugger program; and

under the control of said computer system, said MT safe debugger program checking said memory leak status information and reporting said memory locations designated as in said inaccessible state.

7. The method of claim 1 wherein said memory accesses may be either a read access or a write access and wherein said memory status information maintained by said MT safe debugger program comprises allocated states designated as read-only, write-only, and read or write access.

8. A method for memory leak checking of a multi-threaded target program, said method executable on a computer system having a memory, a clock, one or more central processing units (CPUs), an I/O device for receiving inputs, an I/O device for sending outputs to, and at least one peripheral device, said computer system having

program machine instructions in said memory, said computer system also having a multi-threaded operating system, said method comprising the steps of:

providing a multi-threaded safe ("MT safe") debugger program having memory leak checking facilities, which can operate in conjunction with said multi--threaded operating system;

providing memory leak status information for memory locations in said memory, said memory leak status information indicating at least whether a memory location is in an inaccessible state or not wherein said inaccessible state corresponds to a memory location which is in an allocated state but which is inaccessible in said computer program, said leak status information being maintained by said MT safe debugger program; and

under the control of said computer system, checking said memory leak status information.

10. A computer system for memory access checking of a multi-threaded target program, one or more central processing units (CPUs) and having program machine said computer system comprising:

a memory;

a clock;

at least one central processing unit;

a plurality of program machine instructions loaded into said memory;

a multi-threaded operating system loaded into said memory;

at least one I/O device for receiving inputs;

at least one I/O device for sending outputs to;

at least one of peripheral device;

a multi-threaded safe ("MT safe") debugger program having memory access checking facilities, loaded into said memory and coupled to said multi-threaded operating system;

a multi-threaded target program loaded into said memory under control of said MT safe debugger;

one of said one or more CPUs for executing said multi-threaded operating system and said MT safe debugger to test said multi-threaded target program, said MT safe debugger having a first machine executable mechanism which provides memory status information for memory locations in said memory, said memory status information indicating at least whether a memory location is in an allocated state or in an unallocated state, wherein said allocated state corresponds to a memory location allocated by a computer program and said unallocated state corresponds to a memory location not allocated by said computer program, said status information being maintained by said MT safe debugger program during said test of said multi-threaded target program; and

said MT safe debugger having a second machine executable mechanism which checks said memory status information for each memory location accessed by said multi-threaded target program.

14. The computer system of claim 10 wherein said memory accesses may be either a read access or a write access and wherein said memory status information maintained by said MT safe debugger program comprises allocated states designated as read-only, write-only, and read or write access.

16. The computer system of claim 10 further comprising:

a third machine executable mechanism coupled to said MT safe debugger which provides memory leak status information for memory locations in said memory, said memory leak status information indicating at least whether a memory location is in an inaccessible state or not wherein said inaccessible state corresponds to a memory location which is in an allocated state but which is inaccessible in said computer program, said memory leak status information being maintained by said MT safe debugger program; and

a fourth machine executable mechanism coupled to said MT safe debugger which checks said memory leak status information.

18. A computer system, comprising:

a memory;

a clock;

at least one central processing unit;

a plurality of program machine instructions loaded into said memory;

a multi-threaded operating system loaded into said memory;

at least one I/O device for receiving inputs;

at least one I/O device for sending outputs to;

at least one peripheral device;

a multi--threaded safe ("MT safe") debugger program having memory leak checking facilities, loaded into said memory and coupled to said multi-threaded operating system;

a multi-threaded target program loaded into said memory under control of said MT safe debugger;

one of said one or more CPUs for executing said multi-threaded operating system and said MT safe debugger to test said multi-threaded target program, said MT safe debugger having a first machine executable mechanism which provides memory leak status information for memory locations in said memory, said memory leak status information indicating at least whether a memory location is in an inaccessible state or not wherein said inaccessible state corresponds to a memory location which is in an allocated state but which is inaccessible in said computer program, said leak status information being maintained by said MT safe debugger program during said test of said multi-threaded target program; and said MT safe debugger having a second machine executable mechanism which checks said memory leak status information for each memory location accessed by said multi-threaded target program.

22. A debugger in a computer readable medium for providing a multithreaded safe ("MT safe") mechanism for run-time-checking ("RTC") a multi-threaded target program, said debugger operating in a computer system having a memory, a clock, one or more central processing units (CPUs), an I/O device for receiving inputs, an I/O device for sending outputs to, and at least one peripheral device, said debugger comprising:

a first machine executable structure for maintaining status of memory locations in a computer system, said memory status information indicating at least whether a memory location is in an allocated state or in an unallocated state, wherein said allocated state corresponds to a memory location allocated by a computer program and said unallocated state corresponds to a memory location not allocated by said computer program, said status information being maintained by said MT safe mechanism during a test of said multi-threaded target program; and

a second machine executable structure which checks said memory status information for each memory location accessed by said multi-threaded target program.

23. A debugger as articulated in claim 22 wherein said second machine executable structure performs said checks of said memory status information for each memory location accessed by said multi-threaded target program on a per-thread basis.

24. A debugger as articulated in claim 22 further comprising a reporting mechanism for reporting an error if said status information for said memory location accessed

indicates an unallocated state, said second machine executable structure under control of said MT safe mechanism.

25. A debugger as articulated in claim 24 wherein said reporting mechanism performs said reporting of said memory status information for each memory location accessed by said multi-threaded target program on a per-thread basis.